SIMULATION MODEL OF GROWTH AND DEVELOPMENT OF SWINE

By

FLÁVIO BELLO FIALHO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1997

To my grandfather Clóvis de Oliveira Bello, in memory.

ACKNOWLEDGEMENTS

I would like to deeply thank Dr. Ray Bucklin, my advisor, for all the support that he gave me throughout these four years of my life. It was both an honor and a pleasure to have conducted this work under his orientation. His guidance, dedication and enthusiasm were always encouraging and motivating.

I thank EMBRAPA, for giving me the opportunity to come to the University of Florida, and CNPq, for sponsoring my studies. In particular, I would like to thank Jessy Alves Pinheiro and Carlos Cláudio Perdomo, who gave me all the support I needed from Brazil to carry on my studies. I would also like to thank Irenilza Nääs, without whom I would not have been here.

I appreciate the help of the members of my supervisory committee, Fedro Zazueta, Bob Myer, Doug Dankel and Roger Nordstedt, and of all the staff and faculty of the Agricultural and Biological Engineering Department, specially Bob Tonkinson, Khe V. Chau, Direlle Baird and Pierce Jones. I acknowledge the help of Tom Bridges and Jack Nienaber for supplying part of the data used in the validation of the model. I also thank Phil Fowler and Sencer Yeralan for taking me to the world of microcontrollers, and all the people from the Travel and Recreation Program for taking me outdoors.

To K!m & Dri and all my friends, to my father Jorge Olavo de Paula Fialho, my sister Carla, my brother Lenny, Vó Irene and all my family, I want to express my gratitude for all the love and affection. Finally, I could never thank enough my mother Teresinha da

Silva Bello for the constant incentive, love, guidance, affection and devotion, and for teaching me that a good education is the greatest richness a person can have.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School of the
University of Florida in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

SIMULATION MODEL OF GROWTH AND DEVELOPMENT
OF SWINE

By

Flávio Bello Fialho

August 1997

Chairman: Dr. Ray A. Bucklin
Major Department: Agricultural and Biological Engineering

A simulation model was developed to predict body weight and carcass composition in growing swine. The model is subdivided into modules, each responsible for simulating one portion of the model, such that the modules are as independent of each other as possible. Four modules are defined for simulating the microenvironment surrounding the pig, the feeding program used, the pig's heat balance and its metabolism. Fundamental concepts used in the model include a pool of readily available nutrients, separation of lean and fat tissue deposition, and the use of body temperature as a measurement of the thermal environment. The model predicts weight gain, feed consumption, carcass composition, heat production and body temperature. A graphical user interface program was developed to implement the model in a personal computer environment. The user can set the parameters for the simulation and observe the results in 15 different types of graphs. An expert system is interfaced with the simulation model to analyze the

production system based on the simulation results. After the simulation model runs, the expert system is called and the resulting output is displayed on the screen. The model was calibrated and validated against experimental results, and responds to changes in environmental and nutritional factors in a manner similar to the way a real pig would. Although more research is needed to determine all the model parameters for modern pigs, the model provides a frame onto which the results of such research can be readily incorporated. The expert system interface provides an additional tool for management of swine production systems not found in other simulation models and is flexible enough to easily accept the incorporation of new knowledge as it becomes available.

CHAPTER 1
INTRODUCTION

Swine production has greatly increased in efficiency over the years. Methods to achieve further improvements will require more elaborate analysis of production systems than in the past. The interactions between all the factors that affect swine performance must be taken into account when making decisions, and their result is not always obvious. Due to their dynamic nature, it is impossible to recommend a single production strategy as the most efficient for swine production systems. The use of computers to simulate the outcome of different managing strategies can save much time and enables producers to select the proper strategy more easily. In the near future, simulation models will be used routinely in agriculture, as a mean of obtaining predictions of the consequences of changes in biological, management and economic factors (Pomar *et al.*, 1991c). Developing simulation models has therefore become a priority area of research.

Modeling has other virtues as well. In the process of developing a model, it is possible to gain a better understanding of the system and its interactions. Unknown relationships can be identified and research can be directed toward filling the existing gaps in knowledge. Simulated experiments can be made and the results analyzed before spending time and resources on a real experiment. Different experimental designs can be simulated in order to choose the one that will maximize the information that can be extracted from the real experiment.

Two approaches to developing a simulation model are possible. One is to model one specific portion of the whole process in detail. Once this is accomplished, it is possible to move on to another portion, related to the previous one, and model that portion in detail, and so on. Some problems with this approach quickly come to mind. The usefulness of having only a partial model may be limited, even if it is working perfectly. It is also easy to lose track of which portions of the general model are most important, so one may end up spending a lot of time on a part of the model that does not influence the general output much. Another problem is that when other segments of the model are developed, it might be necessary to change the working parts developed previously to accommodate the new segments.

A more efficient approach in terms of developing a complete model is to outline the general model in less detail and subdivide it into smaller portions. The general model can define the inputs and outputs of each portion and all the relationships that exist between them. Once this is done, each portion can be worked on individually, possibly being subdivided into smaller portions and so on. This way the general model is developed at once, and many of the problems outlined previously can be solved. The problem with this approach is that it is more difficult to validate the overall concept, given that its components are not well developed. Much of the data generated by the different segments of the model may be incorrect, until that specific segment is refined and tested.

The applicability of simulation models is many times limited due to the difficulty of interpretation of the results. A good model should be able to generate a large amount of data. Selecting which results are more relevant for a particular situation or observing the relationships between output and input may require some amount of experience.

Incorporating some sort of interface that is able to interpret the data would broaden the use of the model. The inclusion of an expert system connected to the simulation model permits exactly this kind of interpretation to be done automatically. A knowledge base can be defined with the help of experts in the field being simulated, and some useful advice can be associated with the results of each simulation.

The objective of this dissertation was to develop a simulation model that predicts the growth and development of swine in response to nutritional and environmental factors and to interface it with an expert system. Given the complexity of the problem, there was no intention to completely cover every possible factor involved in the modeled system. Instead, the objective was to structure a frame which outlines the basic principles and subdivides the problem into smaller modules, which can later be refined. The model focused mainly on warm weather and on the effects of heat stress on swine production, although provisions were made to handle cold weather as well.

In order to increase the range of possible users of the model, a software system was developed for use on a personal computer. The programming language chosen to develop the system was C++. The object-oriented nature of this language facilitated the implementation of independent modules and the separation of the knowledge base from the expert system inference engine. The purpose of adding an expert system interface associated with the simulation model was to demonstrate how this could be done. The development of a complete knowledge base will be left as future work.

CHAPTER 2
LITERATURE REVIEW

Heat Transfer and Thermal Regulation

All animals produce heat as a result of metabolic activity. The deep body temperature of homeothermic animals is maintained within narrow limits by a series of thermoregulatory mechanisms, which include physiological and behavioral responses to the environment (Ingram and Mount, 1975). The purpose of these mechanisms is to regulate the rate of heat production and the rate of heat transfer between the animal and its surroundings. In order to maintain homeothermy, heat must be dissipated to the environment at the same rate it is being produced (Bruce and Clark, 1979). Pigs do not sweat in response to an increase in temperature (Ingram, 1974), and growing pigs have relatively little fur and thermal insulation. As such, pigs tend to respond more to changes in the thermal environment than other domestic animals (Mount, 1975), although this may be compensated by behavioral responses such as huddling and wallowing.

Heat transfer can be divided into sensible and latent heat. Sensible heat involves direct heat exchange with the surroundings by conduction, convection or radiation, and depends on the existence of a temperature gradient. Latent heat consists in the evaporation of water at the surface of the skin or inside the respiratory tract, which uses heat to change the enthalpy of the evaporating water without changing its temperature. In

a cold environment, heat transfer is primarily sensible, and the animal attempts to maintain homeothermy by reducing the rate of heat loss and increasing the rate of heat production. In a hot environment, sensible heat transfer is small, and the animal needs to increase latent heat loss in order to keep its body temperature approximately constant (Ingram and Mount, 1975). Close and Mount (1978) verified that at lower temperatures sensible heat loss is large and directly related to feed intake, while at higher temperatures evaporative heat loss is large and directly related to feed intake. The intermediate range of environmental conditions is called the thermoneutral zone, in which both heat production and evaporative heat loss are at a minimum (Ingram and Mount, 1975). Within the thermoneutral zone, pigs are able to maintain heat production approximately constant for a given energy intake (Bruce and Clark, 1979).

The rate of sensible heat loss depends on the temperature gradient between the animal's skin and its surroundings. Skin temperature can be modified by changes in the rate of peripheral blood flow, which affects the tissue's thermal insulation. In a cold environment, the blood vessels in the skin contract, reducing blood flow to a minimum, thus increasing the thermal resistance of the skin. In a hot environment, vasodilatation occurs, and the blood flow transports heat from the inner core of the animal to the skin, facilitating heat loss (Ingram and Mount, 1975).

The thermoregulatory mechanisms attempt to maintain the temperature of the body core stable. In order to achieve that, however, the temperature of the peripheral tissue may vary considerably, causing mean body temperature to fluctuate. The proportion of the animal's body that is at core temperature may vary, depending on the environmental conditions. As a consequence of vasoconstriction, the temperature of the extremities and

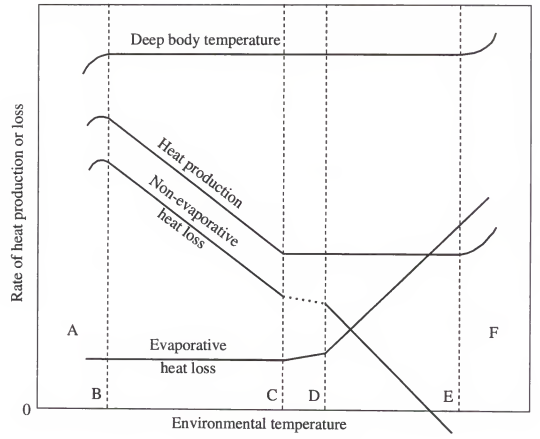
the skin of the animal are more influenced by the external temperature in a cold environment. If ambient temperature is high, vasodilatation increases peripheral blood flow, making the temperature distribution throughout the animal's body more uniform (Ingram and Mount, 1975).

Deep body temperature can be kept stable within a wide range of environments. However, if the environmental conditions are extreme to the point of causing core temperature changes, the animal loses its thermoregulation abilities. For every 10 °C of increase in temperature, the rate of chemical reactions usually doubles (Ingram and Mount, 1975). Thus, an increase in deep body temperature will increase basal metabolism, while a decrease in deep body temperature reduces it.

Usually the animal reacts to a colder environment by increasing basal metabolism in order to generate more heat. However, if the metabolic rate is already at the maximum and the animal is still losing more heat than it is producing, body core temperature will drop at a relatively fast rate. If these environmental conditions persist for a significant amount of time, death by hypothermia occurs. On the other extreme, if the animal is not able to dissipate heat as fast as it is being produced, body core temperature will rise, which causes further increases in heat production due to the increase in the rate of chemical reactions. If this situation persists for a long time, the animal may die from hyperthermia (Ingram and Mount, 1975).

If all the other parameters were kept constant and only environmental temperature changed, the response of the animal would be similar to what is represented in Figure 1 (Mount, 1974). Within the thermoneutral zone, called by Mount (1974) the 'zone of least

thermoregulatory effort', heat production and evaporative heat loss are minimum, and the animal is able to regulate heat loss by changes in posture and peripheral blood flow.



A: zone of hypothermia; B: temperature of summit metabolism and incipient hypothermia; C: critical temperature; D: temperature of marked increase in evaporative loss; E: temperature of incipient hyperthermal rise; F: zone of hyperthermia; CD: zone of least thermoregulatory effort; CE: zone of minimal metabolism; BE: thermoregulatory range.

Source: Mount (1974).

Figure 1. Diagrammatic representation of relationships between heat production, evaporative and non-evaporative heat loss and deep body temperature in a homeothermic animal.

Throughout this dissertation, points C, D and E on the figure are called 'lower critical temperature', 'evaporative critical temperature' and 'upper critical temperature'. The lower critical temperature is also known in the literature as simply 'critical temperature' (Mount, 1974).

As the temperature drops, sensible (non-evaporative) heat losses increase, due to the greater temperature gradient between the animal and the environment. The animal responds to this by increasing heat production, through shivering and other mechanisms (Black *et al.*, 1986). If temperature drops below the lower critical temperature, the animal cannot increase heat production further, and deep body temperature drops. The decrease in temperature causes a reduction in the speed of chemical reactions which decreases heat production (Ingram and Mount, 1975).

Above the thermoneutral zone, sensible heat losses decrease as ambient temperature approaches body core temperature. Animals compensate by increasing latent (evaporative) heat losses, either by increasing the breathing rate, by sweating and increasing the rate of water diffusion through the skin (which is not a very efficient method in the case of pigs), or by purposely wetting the skin using some external source of water. The increased breathing increases energy consumption and reduces the efficiency of use of metabolizable energy (Black *et al.*, 1986). If the temperature rises above the upper critical temperature, when the animal is not able to increase evaporative heat losses further, deep body temperature increases. This causes an increase in the speed of chemical reactions, increasing metabolism and heat production (Ingram and Mount, 1975).

The thermal comfort of an animal cannot be determined by air temperature alone, unless all other conditions are standardized (Mount, 1975). Many factors influence the range of air temperatures perceived by an animal as comfortable. The thermal environment is composed of the various environmental factors that affect heat transfer between the animal and its surroundings. It is not fixed, in the sense that the perception of the environment varies from animal to animal.

In general, smaller and younger animals lose heat more easily than larger and older animals, due to the greater surface to volume ratio and to the smaller proportion of body fat. Newborn pigs may be cold in an environment where mature adults would be hot. Increased air movement will increase conductive heat loss and make the animal feel cooler, while the presence of a radiant heater will result in heat gain by radiation and make the animal feel warmer. The thermal insulation of the floor also affects the rate of heat loss by conduction. Thus, at a given air temperature, the same animal may be hot, if it is exposed to solar radiation, or cold, if it is wet, in the wind and in the shade. In addition, the number of animals in the pen affects the contact area when the animals are huddling, making them more resistant to cold in a larger group. The level of feed intake influences the metabolic heat production, so that the same animal in the same environment may respond differently, depending on the feeding program used (Ingram and Mount, 1975).

All these factors will shift the thermoneutral zone to higher or lower air temperatures. It is therefore impractical to try to express the thermal environment zones as a function of air temperature. In order to correctly evaluate the effect of the environment on an animal, one needs to consider all the environmental factors, as well as

characteristics of the animal and peculiarities of the production system. The thermal environment can be characterized by air temperature, radiant temperature, air speed, humidity, wetting, incidence of solar radiation, temperature and nature of the floor, number of animals in the pen, and temperature of feed and water (Ingram and Mount, 1975).

Although body core temperature is usually maintained at a fairly constant level, this level may vary, depending on the environmental conditions and on the animal's metabolic activity. Close and Mount (1978) measured rectal temperatures varying between 38.7 and 40.2 °C. In order to maintain a stable temperature, the animal must regulate heat production and heat loss as much as it can. As mentioned before, heat transfer can be sensible or latent. The channels through which the animal can exchange heat with the environment are evaporation, radiation, convection and conduction (Mount, 1975).

The rate of sensible heat loss depends on the temperature gradient between the animal and its environment. Sensible heat can be transferred by conduction, convection and radiation. Some amount of heat is transferred to ingested feed and water. Specifically, water intake increases under hot conditions, above what is necessary for the increase in evaporative heat loss (Ingram and Mount, 1975).

Heat is lost by conduction when the animal is in contact with a surface, such as the floor of the pen, which is cooler than body core temperature. The amount of heat lost depends on the temperature of the floor and on the thermal conductance and thermal capacity of the floor material. An animal loses heat faster if it is lying on a cold cement floor, for example, than on a layer of straw bedding (Ingram and Mount, 1975).

Convection is the transfer of heat by movement of air. Warm air moves away from the pig's skin and is replaced by cooler air from the surrounding environment. Heat lost from the skin elevates the temperature of this air to skin temperature. Air movement occurs either by buoyant forces, in which the heated air becomes less dense and rises, or by an external force, such as wind or air movement caused by fans. Air speed and body size directly affect the amount of heat lost by convection (Ingram and Mount, 1975).

The animal can lose heat by long wave radiation to its surroundings, if the temperature of the skin is higher than that of the surrounding surfaces. The amount of heat lost by radiation is proportional to the fourth power of the temperature difference between the surfaces, and is affected by the emissivity of the surfaces. If the surfaces are warmer than the animal's skin, heat will be gained by the animal. This is common if, for example, there is a radiant heat source present. Another possible source of radiation, from which the animal may gain a very large amount of heat, is the sun. Solar radiation has a significantly shorter wavelength than radiation from any surface which would normally surround a pig, and it is usually much more intense. The reflectivity of skin varies according to the wavelength, so it must be evaluated separately for short and long wave radiation (Ingram and Mount, 1975).

As environmental temperature increases, the efficiency of sensible heat loss is reduced, due to the smaller temperature gradient between the animal's skin and its surroundings. The animal can compensate to some degree by vasodilatation, which increases peripheral blood flow and skin temperature. However, this mechanism is limited, and as temperature rises the animal must rely on an increase in evaporative heat loss to compensate for the reduction in sensible heat loss (Ingram and Mount, 1975).

Evaporation of water can occur on the surface of the skin or inside the respiratory tract. Despite the lack of functional sweat glands (the snout is the only place where the pig has sweat glands that respond to temperature), the pig loses some water through the skin by diffusion. Under heat stress, pigs easily become hyperthermic, unless they have access to some sort of water (sprays, mud, etc.) with which they can wet their body surface. If possible, pigs will wallow under heat stress, and this can be the most significant form of heat loss in a hot environment. The rate of evaporation depends on the vapor pressure difference between the skin and ambient air, and also on factors affecting convection, such as air speed and body size (Ingram and Mount, 1975).

When the animal breathes, the inhaled air is heated to body core temperature and humidified to saturation. This process removes heat from the animal's body. When the air is exhaled, it is partially cooled again, which reduces the saturation vapor pressure and causes some of the water to condense inside the respiratory tract. The net amount of heat lost depends on the difference in enthalpy between the inhaled and exhaled air and on the ventilation rate. The heat lost by respiration is part sensible and part latent, although much more heat is usually lost by evaporation of water than by heating of the air (Ingram and Mount, 1975).

Swine Physiology

In order to survive, an animal must eat. Feed is one of the main components of the production cost of swine, so the amount of feed intake per unit of weight gain is a variable of key importance to producers. In normal conditions, there is a direct relation between feed consumption and growth. The exact direction of the cause-effect

relationship is not very clear. On one end, the more an animal eats, the more it will grow. On the other, the fact that an animal is growing will make it utilize the available nutrients more rapidly, and consequently it will eat more. According to Schinckel and Lange (1996), the energy content in the diet and body weight are the two main factors determining voluntary feed intake.

Black *et al.* (1986) assumed that feed intake was determined by the capacity of the animal to utilize nutrients, although it may be limited by gut capacity and inhibited by heat stress. Bridges *et al.* (1992a) made similar assumptions, stating that feed intake is controlled by three main physiological factors: physical fill of the digestive tract, energy density in the blood nutrient pools and heat stress. If the digestive tract is full, the animal cannot physically continue to eat and feed intake will cease. The energy density in the blood (total energy of the nutrients dissolved in the blood divided by total blood volume) will control when the animal starts and stops eating. If the blood energy drops below a certain threshold, the animal feels hungry and begins to eat, in an attempt to replenish the nutrient supply. When blood energy rises, the stimulus to eat ceases and the animal stops eating. The capacity of the animal to utilize nutrients determines how fast blood energy fluctuates, which essentially makes these two factors equivalent. Heat stress reduces or, depending on the degree of stress, completely halts feed intake, as a defense mechanism to decrease heat production. Although the animal still generates heat by maintenance metabolism, and possibly by some tissue catabolism, the heat generated in digestion and tissue deposition is reduced. Under cold stress, feed intake increases to compensate for the increase in the rate of heat loss. Close and Mount (1978) verified an increase in *ad lib.* feed intake with a decrease in air temperature below the thermoneutral zone.

Bridges *et al.* (1992a) modeled the digestion process by dividing the digestive tract into three compartments (stomach, small intestine and cecum-colon), and simulating nutrient absorption and outflow from each of the three compartments. Usry *et al.* (1991) described a detailed model of intestinal tract mechanics in a growing pig. Bastianelli *et al.* (1996) modeled digestion and nutrient absorption in pigs, using a detailed model that divided the digestive tract into stomach, two segments of the small intestine and large intestine. Within each of these compartments, they defined sub-compartments for the different nutrients, in a very detailed model of digestion.

McDonald *et al.* (1991) modeled the eating behavior of growing pigs using a Markov chain to describe the occurrence of eating events. Later, the same model was expanded into a more complex one that could be applied to grouped-penned pigs (McDonald and Nienaber, 1994). Nienaber *et al.* (1990a) measured feeding rates and meal pattern and frequency in pigs, noting an increase in feed intake with a reduction in ambient temperature and verifying that most of the feeding activity occurs during the daylight hours.

Most models which simulate metabolism considered the maintenance requirements to be first on the list of requirements to be met, followed by protein growth and, finally fat growth (Pomar *et al.*, 1991a). Tissue deposition only occurs if there are sufficient nutrients available to promote growth after the maintenance requirements are met. If the readily available nutrients are not sufficient to account for the maintenance requirements, body tissue must be catabolized to supply the necessary nutrients. When using a physiological age concept (Loewer, 1987), tissue catabolism may cause a reduction in physiological age and in the degree of maturity of the animal (Bridges *et al.*, 1992a).

The maintenance requirements traditionally have been expressed as a function of metabolic weight, that is, weight to the power 0.75 (Whittemore and Fawcett, 1974; Bruce and Clark, 1979). Close (1978) calculated a minimum maintenance energy requirement of 440 kJ / $kg^{0.75}$ per day at a thermoneutral air temperature of 25 °C. However, most metabolic activity related to maintenance occurs in the intestines and liver, followed by the muscles, such that the maintenance energy requirements are more a function of protein mass than of total body weight (Whittemore, 1983).

The protein requirements for maintenance and growth are really amino acid requirements. If the amino acid composition of feed protein is close to that required by the animal, the protein may be considered of good quality. Otherwise, the protein intake of the animal has to increase, in order to satisfy the requirement for the most limiting amino acid. The ideal amino acid balance needed for growth, in particular for fast growing animals, should be similar to the balance found in tissue protein (Whittemore, 1983). If the amino acid levels in the nutrient pool are not present in the proper ratio, or they are not available in sufficient quantities, lean tissue will have to be catabolized to satisfy the deficit.

If energy is deficient, the animal must catabolize enough body tissue to supply the necessary energy. Three approaches are possible. One is to first catabolize excess fat tissue, assuming the body first uses the excess tissue, and only catabolize lean tissue if the excess fat tissue is not sufficient to account for the energy deficit (Loewer *et al.*, 1979). Black *et al.* (1986) assumed that any excess or deficiency of metabolizable energy is allocated to or taken from body fat reserves, including the energy needed for protein growth. Using this assumption, the pigs would deposit lean tissue at the expense of fat if

fed a high protein, low energy diet, which is not necessarily true. However, the authors do simulate a reduction in protein deposition associated with a decrease in energy intake.

Another possible approach to model tissue catabolism is to assume that body weight will reduce uniformly, so that both lean and excess fat tissue will be catabolized evenly until the energy deficit is accounted for (Bridges *et al.*, 1992a). The third is an intermediate between the two previous approaches, in which part of the energy comes from excess fat and part comes from a uniform reduction in body weight.

It is generally accepted that there is a linear relationship between rate of energy intake and protein deposition when energy intake is above maintenance but below that required for maximum protein deposition. When the energy supply exceeds the requirements for protein deposition, the excess energy is converted to fat. Animals with reduced feed intake grow slower and allocate a greater portion of the ingested energy and nutrients for maintenance, resulting in poorer feed efficiency. As feed intake increases within the lean tissue deposition range, growth rate increases with only minor changes in the relation between lean gain and fat gain. As intake increases beyond what is necessary for maximum protein growth, the fat : lean ratio in the carcass increases, resulting in a fatter animal (Schinckel and Lange, 1996).

As genetic improvement and better management increase the potential for increased protein deposition, it becomes more likely that energy will become the limiting factor for protein deposition (Schinckel and Lange, 1996). At the same time that this enhances carcass quality, it also imposes a greater challenge for nutritionists and swine farm managers, as they must be able to give the animal the conditions it needs to reach its

potential. Ideally, all the nutrient requirements for maximum lean growth would be satisfied, and any energy deficiency would be reducing the deposition of excess fat.

Bridges *et al.* (1992a) considered the simulated pig to be composed of fat, protein, water and minerals. Fat was divided into 'essential' fat and 'excess' fat. Essential fat and protein were simulated to grow together in a lean mass ratio of 0.4 units of fat per unit of protein. The amounts of water and minerals in the animal's body were considered functions of protein mass.

Close *et al.* (1978) found that environmental temperature affected both protein and fat deposition in pigs, but the effect was greater on fat deposition. As temperature decreased below the thermoneutral zone, fat deposition decreased, due to the higher energy requirement for metabolic heat. The effect was more pronounced at low feed intake levels, due to the smaller base heat production, which shifted the thermoneutral zone to a higher temperature range. As temperature increased above the thermoneutral zone, fat deposition decreased, specially at high feed intake levels. Protein deposition was only slightly influenced by temperature. The level of feed intake influenced both protein and fat deposition, but again fat varied more than protein. This shows how fat deposition is directly affected by the amount of available energy for tissue deposition. Under hot conditions, protein-rich diets are expected to increase heat stress, due to the greater heat increment of protein, relative to fat and carbohydrates.

Close (1978) found that the efficiency of both protein and fat deposition (the increase in tissue deposition per unit increase in metabolizable energy intake) improved with a reduction in temperature. This happened because in a cold environment part of the additional maintenance requirement for supplemental heat was being supplied by the heat

generated by the inefficiencies in the process of tissue deposition. When tissue deposition increased, so did the extra heat generated, reducing the maintenance requirement and apparently increasing the efficiency of tissue deposition.

Bridges *et al.* (1986) defined maximum potential growth curves for protein, fat, ash and water deposition in the animal's body as functions of physiological age. Black *et al.* (1986) defined potential rates of energy and nitrogen deposition as functions of body weight and the currently deposited amounts of energy or nitrogen. They report values for each of these parameters for boars, gilts and castrates from both large and small commercial pig farms in Australia, representing a range of pig genotypes (lean, fast growing and fat, slow growing types).

Models of Swine Growth

There is an abundance of literature about swine in which some sort of treatment is applied and the response is observed in terms of weight gain, feed consumption and feed efficiency or conversion. Though these works have merit, it is simplistic to attempt to understand all the reasons that cause some result to happen based solely on the observation of the result. This can make the results only applicable to conditions similar to those observed in the experiment. In order to be able to apply the results to a wider range of conditions, a large number of experiments must be made.

In an attempt to understand how the underlying physiological principles interact in a number of particular situations, simulation models of swine growth have been developed. These models allow the rapid prediction of the outcome of a number of different

conditions, in order to select the most adequate. Many models of swine physiology and growth exist in the literature, each with its strengths and weaknesses.

Teter *et al.* (1973) described the relation between effective environmental temperature, feed intake and heat production. Bruce and Clark (1979) developed two models for heat production of growing pigs: one for thermoneutral environments and another for cold environments. By combining both models, they were able to determine ambient temperature at the lower limit of the thermoneutral zone, which they called critical temperature. Their model, however, concentrated on cold weather, and does not account for the physiological changes that take place during heat stress.

Whittemore and Fawcett (1974) developed a simple model describing the response of growing pigs to energy and protein intake. They later refined the model (Whittemore and Fawcett, 1976), exploring protein synthesis and breakdown in more detail. Jacobson *et al.* (1989) modified the model to include the heat transfer processes in more detail and used their modified model to predict growth in early-weaned piglets. Whittemore (1983) compiled guidelines for the development of swine metabolism models that deal with energy and protein requirements.

Moughan and Smith (1984) developed a model for predicting daily protein and lipid deposition in swine based on the diet composition and amino acid digestibility. Their model used simple principles and assumed that the digestibility of each amino acid from each ingredient in the feed is known. They considered that the amino acids not used to meet the maintenance requirements are used for protein deposition, limited by the animal's genetic potential. Amino acids that are not used for protein deposition were assumed to be deaminated to supply energy.

A sensitivity analysis was done on the above model (Moughan, 1985).The ideal level of the first limiting amino acid and the maximum genetic potential for protein deposition were found to be the parameters that most influenced the output of the model. The model, however, predicted only instantaneous tissue deposition rate, and not body growth over a period of time. Moughan *et al.* (1987) later expanded the model to simulate growth over the 20 to 90 kg body weight range.

Black *et al.* (1986) simulated the utilization of energy and amino acids in pigs, in a broad model that covered all physiological states, including gestation and lactation. Their model accounted for both environmental and nutritional factors, in both cold and hot environments. However, it simulated growth on a daily basis, and was not capable of detailing the feeding process and heat balance during the course of the day.

Watt *et al.* (1987) described a modularized simulation model called NCCISWINE for swine production systems. They divided the model into three modules, the first responsible for the housing and environment around the pig, the second for nutrition and diets used in the simulation, and the third for pig growth based on protein synthesis.

Stombaugh and Stombaugh (1991) developed a model of protein synthesis and deposition in growing pigs. Their model was based on a rectangular hyperbola equation that calculated protein synthesis as a function of available energy and protein. This function assumes that protein synthesis is proportional to the available protein if the energy supply is infinite and vice-versa. However, it does not seem reasonable that protein deposition will still be correlated to available protein if its amount is, for example, three times the requirement. Furthermore, if the proposed method is expanded to include amino acids, the equation would have to be reformulated in some non-intuitive manner.

Bridges *et al.* (1992a,b) developed a model of the physiological growth of swine, with emphasis on the digestive process. The model, called NCPIG, was a result of the work of the North Central Regional Project Physiological Modeling Group (NC-204), in cooperation with the University of Kentucky and the USDA Meat Animal Research Center. They based their model on the physiological age concept described by Loewer *et al.* (1987). Basically, physiological weight is the minimum weight necessary for a given level of physiological development, and physiological age is the minimum age to achieve that weight. For each physiological weight, there is a minimum amount of 'physiological' water, protein, fat and minerals. 'Excess' material was defined as the mass of each material beyond the physiological minimum.

Loewer *et al.* (1987) assumed that a limited amount of excess protein and minerals and an unlimited amount of excess fat may be stored, but excess water cannot be stored and is eliminated. According to this concept, excess fat is the storage mechanism for the energy intake not used to synthesize other body components. The body does not store large amounts of excess protein, and the total amount of minerals is small compared to other components. Therefore, it is possible to assume that the only component with significant amounts of excess storage is fat. According to Loewer *et al.* (1987), it is possible to have weight gain while the animal is in a negative energy balance, if only energy is deficient. This would be a result of excess fat tissue catabolism and simultaneous lean tissue formation. Since lean tissue contains much more water than fat tissue, a net gain in weight can occur, for a limited time.

Bridges *et al.* (1992a) used the concept of a nutrient pool, to where the absorbed and catabolized nutrients are directed, and from where the nutrients used for maintenance,

growth and other requirements are extracted. The nutrient pool consists of the nutrients which are in a readily available form. The authors assumed that the nutrients in the pool are the ones present in the blood. However, tissues with high metabolic activity, such as the liver and kidneys, also contain a significant amount of nutrients that can be considered readily available. Usry *et al.* (1992) described the simulation model of heat exchange between the pig and its environment used in conjunction with the growth model of Bridges *et al.* (1992a).

Pomar *et al.* (1991a) developed a swine growth model in which they considered not only protein and fat, but also body DNA accumulation. However, their model did not account for effects of the environment. The same authors expanded the model (Pomar *et al.*, 1991b) to include the reproductive cycle (reproduction, gestation and lactation). A dynamic herd model was also developed (Pomar *et al.*, 1991c), which included stochastic simulation of some of the model parameters.

Vries and Kanis (1992) modeled the economic returns of varying *ad lib.* feed intake capacity, minimum fat to protein deposition ratio, maximum protein deposition rate and feed intake in pigs. They concluded that the maximum economic gain was achieved when feed intake was just enough to satisfy the maximum protein deposition rate. The physiological model used was very simple, and assumed that energy was the only limiting factor for both protein and fat deposition.

Schinckel and Lange (1996) discussed the growth parameters needed for pig growth simulation models and listed what they considered the most important. These were daily whole-body protein accretion potential, partitioning of energy intake over maintenance

between protein and lipid accretion, maintenance requirements for energy and daily feed intake.

In a completely different approach to modeling, Korthals *et al.* (1994) developed and trained a set of neural networks to estimate daily weight gains as a function of varying temperature. They selected the neural net with the best results, but it still overestimated growth rate. The authors believe that training the net with a more extensive set of data would improve the predictions.

Ventilation in housing systems for swine have also been modeled (Nääs *et al.*, 1990; Bucklin *et al.*, 1991). In this model, the focus was not on the pig, but on removing heat and humidity from the building used to house the animals. Axaopoulus *et al.* (1992) developed a simulation model to evaluate the microenvironment around pigs under hot conditions.

Expert systems have not been common in the literature about swine. Jones *et al.* (1987) developed an expert system to diagnose environmental problems with confinement swine buildings. The expert system consisted of 33 rules that dealt with ventilation problems. There was no simulation model of any type associated with the expert system.

There are obviously many models of swine growth in the literature. Some are more detailed than others and some focus on one specific aspect of production, while others are broader. However, none of the existing models propose a simple way to interface with less knowledgeable users. The present study fills this gap by building a comprehensive model and complementing it with an expert system interface.

CHAPTER 3
SYSTEM OVERVIEW

The purpose of this study was to develop a computer system to evaluate and optimize the environment and nutrition program of a swine producing facility. The system is divided into two main parts, as shown in Figure 2. The first part is a simulation model of the growth and development of swine. This model is designed in a modular fashion, in which each module is responsible for simulating a piece of the whole system. The modules are designed to be as independent of each other as possible, although some amount of information exchange between the modules is always necessary. The second part is an expert system which suggests improvements to optimize the facility. It processes the simulation output using a predefined set of rules and suggests modifications to the system, which may or may not be accepted by the user. After the expert system has executed, the simulation may be run again and the results compared with the previous ones.

The objective of developing a computer program was to create a simple interface for the simulation model, suitable for use as a management tool. The program was developed for use in a personal computer running a commonly used operating system. The programming language chosen was C++, due to its object-oriented nature. The separation of the model into different modules makes an object-oriented language suitable for implementing the system. Each module is defined as a class, which contains all the

module's parameters, as well as functions to initialize the simulation and to simulate a time step. A main simulation class calls the simulation functions of each module. The use of C++ also permitted the expert system knowledge base to be separated from the inference engine and allowed the development of a simple, yet very efficient inference engine.

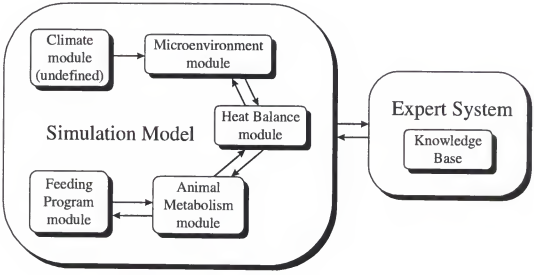


Figure 2. Overview of the program structure.

Simulation Model

The simulation model is divided into interconnected modules. The main modules are the Animal Metabolism module and the Heat Balance module. Other modules defined are the Feeding Program module, the Microenvironment module and the Climate module. The Climate module was not implemented in this study. Its purpose is to generate weather data outside the building. When developed, it should supply the Microenvironment

module with the current values of temperature, humidity, wind speed and direction, solar radiation, position of the sun, rainfall, and any other climatic variables of importance.

The Microenvironment module is responsible for simulating the environment surrounding the animal. It generates data for air temperature, radiant temperature, floor temperature, solar radiation, humidity, air pressure, air speed, thermal conductance of the floor, number of animals in each pen, availability of shade, and temperature of ingested feed and water. This data is supplied to the Heat Balance module, which returns the heat and moisture loss from each animal. The Microenvironment module is also responsible for everything concerning the thermal behavior of the building surrounding the animal, including simulation of the heat balance of the building. In the present study, this function was not implemented, and the module's sole purpose is to supply information to the Heat Balance module. In the future, different versions of the Microenvironment module can be developed for different types of buildings.

The Heat Balance module simulates the animal's body temperature, as well as heat and moisture losses to the environment. It interfaces with the Microenvironment module as described earlier. It also supplies the Metabolism module with body temperature, and receives back information on the animal's total heat production, feed intake, body weight and the animal's specific heat. The fundamental assumption made in this module is that the animal's reaction to the environment is a function of body temperature.

The Feeding Program module is responsible for controlling what type and how much feed will be supplied at which time of day, or if feeding is *ad lib.*, based on a feeding strategy defined by the user. It supplies the Animal Metabolism module with data on feed composition and the amount of available feed, and receives back data concerning

the amount of feed consumed. The Feeding Program module is also responsible for keeping a record of the amounts of feed consumed and feed wasted.

The Animal Metabolism module uses data from the Feeding Program and Heat Balance modules and simulates feed intake, digestion, maintenance and tissue deposition and catabolism. It is the module that generates the most valuable data for the user (feed consumption, body weight gain, feed efficiency and carcass composition). The fundamental concept developed in this module is that of a pool of readily available nutrients, to where digested and catabolized nutrients are directed, and from where the nutrients required for maintenance and tissue deposition are extracted.

The present study focused on the Animal Metabolism and the Heat Balance modules. The Feeding Program module was also fully developed. Further work will be needed on the Microenvironment and Climate modules. Throughout the description of the model, suggestions for future work are included. As an example, some of the areas that could be improved are the nutrient composition of the nutrient pool (perhaps including individual amino acids and minerals) and the digestion process.

The model is solved numerically, using Euler's first order method. At each time step, the system uses the current values of the model's state variables to estimate the values for the following time step. Where possible, parts of the model were solved analytically, in order to improve the accuracy of the estimates. The model is developed as a series of steps, the order of which is important. The simulation process follows the order in which these steps are described.

Expert System

The expert system uses output from the simulation program to suggest actions which may improve the performance of the swine production facility. The actions suggested by the expert system may or may not be accepted by the user. The expert system contains a set of rules, called the knowledge base, which it uses to test the data base generated as output of the simulation program. The knowledge base currently defined is a very simple one. The idea was to demonstrate how an expert system can be interfaced with the simulation model and function as a tool to optimize a production system.

The user can makes changes to the simulation parameters, based on the recommendations given by the expert system. The simulation model can be run again and the results compared to those obtained previously. This allows the evaluation of the extent to which the modifications suggested by the expert system and accepted by the user were effective. This new simulation produces a new data base, which is again processed by the expert system, to further optimize production. This process can go on until the user is satisfied (or not) with the simulated results.

The knowledge base is structured as a set of rules that act on a set of predefined parameters. These rules have a simple basic structure. They have a premise, which can be true or false, depending on the values of the parameters. They also have an action, which is executed if the premise is true. Parameters are implemented as real numbers, the meaning of which depends on the individual parameter. All parameters are initialized to

invalid values at the beginning of the expert system session, indicating that no information about the parameter is currently available.

As each parameter is modified, the expert system's inference engine tests all the rules that depend on the parameter. If the premise of a rule is found to be true and the rule has not been executed before, the action of the rule is executed. The action of a rule usually modifies the value of one or more parameters, which in turn may trigger other rules. The restriction that a rule can only execute once avoids infinite loops.

When the expert system is executed, the inference engine first resets all the rules (indicating that none of them have been triggered) and parameters, and then starts reading the data base. Each entry in the data base sets the value of a parameter. This may trigger one or more rules that depend on that parameter. After all rules have been tested and all the resulting actions executed, the inference engine reads the next entry from the data base. This process continues until there are no more entries in the data base. At this point, the expert system finishes execution. Each time the system suggests a modification, the user may choose to halt the expert system and go back to the simulation program.

The expert system knowledge base can be expanded in the future. In fact, the current knowledge base is very limited, and should be revised. One of the advantages of the expert system approach is that the knowledge base is flexible, and can be modified as new research results become available. It only needs to be recompiled and the system is ready to run with the new rules.

CHAPTER 4
FEEDING PROGRAM MODULE

One of the most influential factors in swine growth and development is the amount and type of feed consumed by the animal. A feeding program is a method used for supplying feed to the animals during the course of their lives. It specifies, for each day of the animal's life, the type of feed used, the amount fed, the feeding strategy (*ad lib.* or restricted), the time it is supplied, and any restrictions imposed during the course of the day.

In order to characterize feeding programs and how they affect feed consumption, it is necessary to define a few terms used in the model. *Feed supplied* (R_s, kg) is the amount of feed put into the feeder, but not necessarily consumed by the animal. *Feed consumed* (R_c, kg) is the amount of feed actually consumed by the animal. *Feed wasted* (R_w, kg) is the amount of feed that the animal throws away in the process of eating, plus any feed that is discarded from the feeder, for health reasons or management practices. These three variables are cumulative and, in a long term analysis, R_s is equal to the sum of R_c and R_w. *Feed use efficiency factor* (e_R, dimensionless) is the proportion of supplied feed that is actually eaten by the animal (the rest is wasted), and depends on the method of feeding and feeder type. *Feed in the feeder* (R_f, kg) is the total amount of feed present in the feeder. *Available feed* (R_a, kg) is the amount of feed in the feeder that is readily available for the animals to consume. It is equal to R_f multiplied by e_R, when the animal has access

to the feed, or zero when feed is restricted by removing the feeder from the reach of the animals.

Feeding Phases

The feeding program consists of a list of feeding phases. Each phase specifies the diet type, the daily feeding schedule used, the initial daily amount to be fed and a weekly increase in this amount during the phase. *Ad lib.* feeding may be specified, which means there will be no restriction on the amount fed, except for a possible withdrawal of feed during certain periods of the day, defined in the feeding schedule. The transition from one feeding phase to another takes place based on the animal's age and weight. The transition may either occur when both minimum age and minimum weight are satisfied, or when at least one of the two criteria is satisfied. The type of transition used is defined in the feeding program.

Feed Type

Each phase in the feeding program requires the type of feed to be specified. Feed types differ in nutrient composition. To characterize a feed type, it is necessary to specify the amount of each nutrient per unit weight of feed. The model expresses feed composition as a *vector of nutrients in the feed* ($\mathbf{N_f}$), currently defined as:

$$\mathbf{N_f} = \begin{bmatrix} P_f \\ L_f \\ G_f \\ F_f \\ v_f \end{bmatrix} \qquad (1)$$

where P_f, L_f, G_f and F_f are the amount of *protein, fat, carbohydrates and fiber in the feed*, in grams per kg of feed, and v_f is the *specific volume of ingested feed in the digestive tract* (cm^3 / kg), which should not be confused with the specific volume of the feed.

After ingestion, feed absorbs water and occupies a larger volume than when it is dry. The value of v_f should reflect the final volume occupied by the feed within the digestive tract, after it has reached osmotic equilibrium with its surroundings. The volume occupied by feed in the digestive tract will depend on feed composition. Feeds with more fiber tend to be bulkier, while feeds with high water content will not need to be moisturized as much and will occupy less space per unit weight of feed. Parameter v_f relates volume in the digestive tract with feed weight before consumption, and is a function of feed composition. As with the nutrient pool vector $\mathbf{N_p}$, defined in the Animal Metabolism module, $\mathbf{N_f}$ can be refined to include amino acids and other nutrients.

Feeding Schedule

A feeding schedule is a method of managing feed during the course of a day. Each feeding schedule is composed of a list of actions to be taken at certain times of the day, concerning feed supply or restriction. Each action may change the values of available feed (R_a) and feed in the feeder (R_f). The effect of each action depends on whether feeding is *ad lib.* or not. If feeding is *ad lib.*, R_f is set to zero and R_a is set to an arbitrarily large amount. The following actions are currently defined:

"Remove feed": The feed currently in the feeder is taken away from the animals, making R_a equal to zero, although R_f does not change:

$$R_a = 0 \tag{2}$$

"Restore feed": The feed previously removed is returned to the animals, making R_a equal to $R_f \cdot e_R$:

$$R_a = R_f \cdot e_R \tag{3}$$

In case of *ad lib.* feeding, R_a is made arbitrarily large. This also generates a feeding stimulus, which under certain circumstances may induce the animals to start eating.

"Level feed": The amount of feed in the feeder is set to a given level R. If the feed had been removed, it is restored. If the new level is greater than R_f, the difference is recorded as feed supplied (R_s); if it is less than R_f, the difference is discarded and recorded as feed wasted (R_w). The value of R_f is set to the new level of feed in the feeder and R_a is set to $R_f \cdot e_R$:

$$\begin{cases} R_s = R_s + R - R_f & , R_f < R \\ R_w = R_w + R_f - R & , R < R_f \end{cases} \tag{4}$$

$$R_f = R \tag{5}$$

$$R_a = R_f \cdot e_R \tag{6}$$

In case of *ad lib.* feeding, the value of R is irrelevant, no waste or supply of feed is recorded, and this action has the same effect as restoring feed.

"Supply feed": A given amount of feed is added to the feeder, increasing R_f. If the feed had been removed, it is restored, making R_a equal to the $R_f \cdot e_R$. The amount added is recorded as feed supplied (R_s):

$$R_s = R_s + R \tag{7}$$

$$R_f = R_f + R \tag{8}$$

$$R_a = R_f \cdot e_R \tag{9}$$

In case of *ad lib.* feeding, the value of R is irrelevant, no supply of feed is recorded, and this action has the same effect as restoring feed.

"Discard feed": The feed currently in the feeder is discarded, making both R_a and R_f equal to zero. The amount discarded (the old value of R_f) is recorded as feed wasted (R_w):

$$R_w = R_w + R_f \tag{10}$$

$$R_f = 0 \tag{11}$$

$$R_a = 0 \tag{12}$$

In case of *ad lib.* feeding, no waste is recorded when feed is discarded, and this action has the same effect as removing the feed.

At least one action in the feeding schedule must be of type "Level feed" or "Supply feed". The amount of feed is specified as a percentage of daily consumption. The amount in kg is calculated, based on the daily feed consumption specified in the feeding program.

During the simulation procedure, the model compares the current time with the schedule of actions, and modifies R_f and R_a accordingly.

For each time step, the feeding program determines the feed composition and the amount of available feed for each animal and passes this information to the Animal Metabolism module. In case of *ad lib.* feeding, R_a will either be zero (if the feeding schedule imposed feed withdrawal during the time step) or an arbitrarily large number. At the end of the time step, the *feed intake in the time step* (R_i, kg) calculated in the Animal Metabolism module is used to update the values of R_c, R_w, R_f and R_a:

$$R_c = R_c + R_i \tag{13}$$

$$R_w = R_w + \frac{R_i}{e_R} - R_i \tag{14}$$

$$R_f = R_f - \frac{R_i}{e_R} \tag{15}$$

$$R_a = R_a - R_i \tag{16}$$

In case of *ad lib.* feeding, the values of R_f and R_a are not changed, but the amount of feed consumed and wasted is added to the feed supplied:

$$R_s = R_s + \frac{R_i}{e_R} \tag{17}$$

List of Symbols

e_R	feed use efficiency factor
F_f	fiber in the feed (g / kg of feed)
G_f	carbohydrates in the feed (g / kg of feed)
L_f	fat in the feed (g / kg of feed)
$\mathbf{N_f}$	vector of nutrients in the feed
P_f	protein in the feed (g / kg of feed)
R_a	available feed (kg)
R_c	feed consumed (kg)
R_f	feed in the feeder (kg)
R_i	feed intake in the time step (kg)
R_s	feed supplied (kg)
R_w	feed wasted (kg)
v_f	specific volume of ingested feed in the digestive tract (cm^3 / kg)

CHAPTER 5
ANIMAL METABOLISM MODULE

The Animal Metabolism module simulates feed intake and digestion, basal metabolism, tissue deposition and heat production within the pig. The model assumes the existence of a pool of readily available nutrients in the animal's body. Simulation of animal metabolism is based on the flow of nutrients to and from this pool. Nutrients are removed from the pool for maintenance and tissue deposition. The nutrients in the ingested feed are incorporated into the pool after the digestion process, as are the nutrients made available by tissue catabolism.

Overview

The model considers the body to be formed by three components. Lean tissue is formed by protein, nucleic acids, minerals and other components, and consists of the dry, fat-free portion of the animal's body (mainly muscles and bones). A certain amount of energy and an adequate balance of nutrients are required in order for lean tissue growth to occur. Body water includes all the water in the animal's body, and does not require energy nor nutrients to be deposited. The model assumes that the amount of water in the animal's body is proportional to the amount of lean tissue.

Fat tissue consists of the fat deposits in the animal's body. The model assumes that no specific nutrients (only energy) are required for fat deposition. A certain minimum

amount of fat tissue growth, called essential fat, is required for lean growth, and the energy required for essential fat growth is added to the lean tissue growth requirements. The *minimum fat to lean ratio* ($k_{F:L}$) determines the minimum amount of fat which must be formed per unit of lean tissue. Excess fat is formed when there is more available energy than is required for lean tissue growth. This can occur either if feed intake is greater than the maximum lean growth requirements or if nutrient intake is unbalanced and there is a deficiency in one of the nutrients.

The sum of *lean tissue weight* (W_{lt}, kg), *fat tissue weight* (W_{ft}, kg) and body water make up empty body weight. A correction factor must be applied to obtain the animal's *body weight* (W_b, kg):

$$W_b = \frac{W_{lt} \cdot (1 + k_{WT}) + W_{ft}}{k_{EB}} \tag{18}$$

where k_{WT} is the amount of *body water per unit of lean tissue* and k_{EB} is the *empty body weight fraction*, relative to total body weight.

Many parameters of the model can be expressed as functions of body weight. However, two animals with the same weight but different carcass composition do not necessarily behave in the same way. Considering that lean tissue has a more active metabolism than fat tissue, some correction must be made for the amount of fat in the carcass. For example, a fat animal should have a smaller basal maintenance requirement than a lean animal with the same weight. One way to account for that is to define an *equivalent body weight* (W_e, kg) for the animal, based on the amount of lean and fat tissue in the animal's body. W_e is defined as the weight of a lean animal with the same basal metabolism as the animal being evaluated, and is obtained by:

$$W_e = W_b - \frac{(1 - k_F) \cdot W_{ft}}{k_{EB}} \tag{19}$$

where k_F is a dimensionless constant between 0 and 1 which expresses the *lean tissue equivalent of fat tissue*, i. e., the mass of lean tissue with a metabolism equivalent to one mass unit of fat tissue. If a value of 1 is used for k_F, there is no difference between body weight and W_e. On the other extreme, a value of 0 means the effect of fat is completely ignored when W_e is used, and it is the equivalent of using protein mass to determine the rates of metabolic functions. An intermediate value would most likely be correct.

The simulation procedure is carried out in the following order. First, feed intake and feed digestion are calculated, and the resulting digested nutrients are added to the nutrient pool. Second, the animal's maintenance requirements are calculated. If any of the required nutrients is not available in adequate amounts, lean tissue is catabolized, and the nutrients made available are added to the nutrient pool. If there is energy deficiency in the nutrient pool, both lean and fat tissues are catabolized.

Only after the maintenance requirements have been met will there be tissue deposition. Lean tissue is deposited first, together with associated fat, in an amount determined either by the animal's genetic potential or by the most limiting nutrient in the pool, whichever is less. After lean tissue deposition, the model simulates excess fat tissue deposition, which is again limited by the animal's genetic potential and by the amount of energy available in the nutrient pool. It is possible to have lean tissue catabolism and fat tissue deposition during the same time step if, for example, the nutrients in the feed are unbalanced.

The values of some parameters in the model depend on the current value of *body temperature* (T_b, °C), relative to the thermal comfort zones. Under cold stress, when T_b falls below the *lower critical temperature* (T_c, °C), the animal will increase metabolism in order to produce heat, in order to bring T_b back up to T_c. The amount of additional heat than can be generated is limited, and after the maximum is reached the animal will no longer be able to maintain body temperature and will become hypothermic. Under heat stress, when T_b rises above the *evaporative critical temperature* (T_e, °C), the animal will reduce feed intake, in order to decrease energy input and heat production. As T_b approaches the *upper critical temperature* (T_h, °C), feed intake is reduced to zero. If T_b rises above T_h, the animal can no longer regulate heat losses and becomes hyperthermic, in which case T_b will start to rise fast.

Nutrient Pool

The nutrient pool consists of the readily available nutrients temporarily stored in the blood and tissues with high metabolic activity, such as the liver. The nutrients digested and absorbed by the animal, as well as the nutrients originated as a result of tissue catabolism are incorporated into the nutrient pool. The maintenance and productive processes remove nutrients from the pool. The *nutrient pool vector* ($\mathbf{N_p}$) is currently defined as:

$$\mathbf{N_p} = \begin{bmatrix} P_p \\ L_p \\ G_p \end{bmatrix} \tag{20}$$

where P_p, L_p and G_p are the amounts of *protein, fat and carbohydrates in the nutrient pool* (g). Further refinement of the model may increase the number of nutrients in this vector. For example, individual amino acids may be added in order to refine the protein requirement, and minerals and vitamins may also be included.

Different nutrients contain different amounts of energy, and the efficiency of use of this energy can vary. Each nutrient has a certain quantity of metabolizable energy (the energy in the nutrient minus the energy in the products of its decomposition), part of which consists of heat increment (the minimum amount of heat generated in the metabolism of the nutrient). The energy available for maintenance and production is called net energy, which is the metabolizable energy minus the heat increment. The *vector of metabolizable energy* ($\mathbf{E'}_{MN}$) and the *vector of net energy* ($\mathbf{E'}_{NN}$) define the energy contents of each nutrient:

$$\mathbf{E'}_{MN} = \begin{bmatrix} E_{MP} & E_{ML} & E_{MG} \end{bmatrix} \tag{21}$$

$$\mathbf{E'}_{NN} = \begin{bmatrix} E_{NP} & E_{NL} & E_{NG} \end{bmatrix} \tag{22}$$

where E_{MP}, E_{ML} and E_{MG} are the metabolizable energy of protein, fat and carbohydrates (kJ / g) and E_{NP}, E_{NL} and E_{NG} are the net energy of protein, fat and carbohydrates (kJ / g).

Another important variable monitored by the model is the *heat production in the time step* (Q_P, kJ), which is initialized to zero at the beginning of each time step. In order to calculate heat production during tissue deposition and catabolism, it is necessary to know the composition of the tissue. The *vector of lean tissue composition* ($\mathbf{N}_{lt}$) and the *vector of fat tissue composition* ($\mathbf{N}_{ft}$) define the nutrient contents of each tissue:

$$\mathbf{N}_{lt} = \begin{bmatrix} P_{lt} \\ L_{lt} \\ G_{lt} \end{bmatrix} \tag{23}$$

$$\mathbf{N}_{ft} = \begin{bmatrix} P_{ft} \\ L_{ft} \\ G_{ft} \end{bmatrix} \tag{24}$$

where P_{lt}, L_{lt} and G_{lt} are the protein, fat and carbohydrate content of lean tissue (g / kg) and P_{ft}, L_{ft} and G_{ft} are the protein, fat and carbohydrate content of fat tissue (g / kg).

If the nutrient pool vector changes, as the model is refined, each of the vectors described above will also change accordingly.

Feed Intake and Digestion

In each time step, the amount of *available feed* (R_a, kg) may limit feed consumption by the animal. Feed composition is characterized by the *vector of nutrients in the feed* ($\mathbf{N}_f$), which is currently defined as:

$$\mathbf{N}_f = \begin{bmatrix} P_f \\ L_f \\ G_f \\ F_f \\ v_f \end{bmatrix} \tag{25}$$

where P_f, L_f, G_f and F_f are the amount of *protein, fat, carbohydrates and fiber in the feed* (g / kg) and v_f is the *specific volume of ingested feed in the digestive tract* (cm^3 / kg). As mentioned in the description of the Feeding Program module, v_f is the final volume occupied by the feed within the digestive tract, after it has reached osmotic equilibrium

with its surroundings, per kg of ingested feed. Both R_a and $\mathbf{N_f}$ are generated by the Feeding Program module.

When feed is consumed by the animals, it is temporarily stored inside the digestive tract, while it is being digested and absorbed into the animal's body. A *vector of nutrients in the digestive tract* ($\mathbf{N_i}$) is defined, to represent the nutrients that were ingested but not yet absorbed. This vector has a structure similar to $\mathbf{N_f}$:

$$\mathbf{N_i} = \begin{bmatrix} P_i \\ L_i \\ G_i \\ F_i \\ V_i \end{bmatrix} \tag{26}$$

where P_i, L_i, G_i and F_i are the amount of protein, fat, carbohydrates and fiber in the digestive tract (g), and V_i is the volume of feed in the digestive tract (cm^3).

Feed Intake

The amount of feed consumed by one animal depends on a variety of factors. The model assumes that one of the main mechanisms that determines voluntary feed consumption is the *energy concentration in the nutrient pool* (E_c, kJ / kg), calculated as a function of the amount of metabolizable energy in the nutrient pool and equivalent body weight:

$$E_c = \frac{\mathbf{E'_{MN}} \cdot \mathbf{N_p}}{W_e} \tag{27}$$

If E_c at the beginning of the time step falls below a certain *minimum nutrient pool energy threshold* E_{min}, a signal is given to initiate feed consumption. The animal will then begin to eat, and will continue to do so until E_c rises above a *maximum nutrient pool*

energy threshold E_{max}, when the animal loses interest for the feed. Under heat stress, animals decrease voluntary feed consumption. The values of E_{min} and E_{max} are maximum in a thermoneutral or cold environment, being reduced by an increase in body temperature above T_e. If T_b rises above T_h, feed intake stops completely and the two parameters become zero:

$$\begin{cases} E_{min} = E'_{min} & , T_b \le T_e \\ E_{min} = E'_{min} \cdot \left(\dfrac{T_h - T_b}{T_h - T_e} \right) & , T_e < T_b < T_h \\ E_{min} = 0 & , T_h \le T_b \end{cases} \quad (28)$$

$$\begin{cases} E_{max} = E'_{max} & , T_b \le T_e \\ E_{max} = E'_{max} \cdot \left(\dfrac{T_h - T_b}{T_h - T_e} \right) & , T_e < T_b < T_h \\ E_{max} = 0 & , T_h \le T_b \end{cases} \quad (29)$$

where E'_{min} and E'_{max} are the thermoneutral values for E_{min} and E_{max}.

There are behavioral aspects to be considered when simulating feed intake. Pigs are curious animals by nature, and if a pig sees feed being delivered it will most likely try to eat it. In order to account for that behavior, the model may fire a *feeding stimulus* (X_F) whenever feed is supplied. Variable X_F is of type Boolean and it is true if feeding was stimulated in the time step, and false otherwise. If X_F is true and E_c is smaller than E_{max}, the animals will start eating. This behavior might be desirable if the objective is to induce the animals to feed during the coolest times of the day, in order to avoid heat stress.

Once feeding starts, the *feed intake in the time step* (R_i, kg) depends on how fast the animal eats. In general, larger animals eat faster and are able to eat more than smaller ones, due to the larger capacity of the digestive tract. Also, animals with empty stomachs will eat faster than animals that are nearly full. It is assumed that feed intake is

proportional to the difference between V_i and the *maximum capacity of the digestive tract* (V_{max}, cm^3), relative to V_{max}. The capacity of the digestive tract is assumed to be proportional to the equivalent metabolic body weight:

$$V_{max} = k_v \cdot W_e^{0.75} \tag{30}$$

where k_v is the *relative capacity of the digestive tract* ($cm^3 / kg^{0.75}$). The digestion process removes material from the digestive tract, thus reducing V_i and influencing feed intake. The digestion of feed is assumed to be proportional to the amount of material in the digestive tract.

The *rate of feed intake* (r_i, h^{-1}) is defined as the feed intake per unit of time as a fraction of the available capacity of the digestive tract (V_{max} minus V_i), and the *rate of digestion* (r_d, h^{-1}) is the fraction of V_i that is digested per unit of time. Since intake and digestion occur simultaneously, it is convenient to express the change in V_i as a differential equation:

$$\frac{dV_i(t)}{dt} = r_i \cdot \left(V_{max} - V_i(t)\right) - r_d \cdot V_i(t) \tag{31}$$

which, when solved, yields:

$$V_i(t) = V_i(0) \cdot e^{-(r_i + r_d)t} + \frac{r_i}{r_i + r_d} \cdot V_{max} \cdot \left(1 - e^{-(r_i + r_d)t}\right) \tag{32}$$

The value of r_i is assumed to be affected by heat stress, being maximum for $T_b \leq T_e$ and zero for $T_b \geq T_h$:

$$\begin{cases} r_i = r_i' & ,T_b \leq T_e \\ r_i = r_i' \cdot \left(\dfrac{T_h - T_b}{T_h - T_e}\right) & ,T_e < T_b < T_h \\ r_i = 0 & ,T_h \leq T_b \end{cases} \tag{33}$$

where r'_i is the *thermoneutral value for* r_i. If the model determined that the animal is not eating in the time step, or if R_a is zero, r_i is set to zero and, consequently, R_i will be zero.

The variables needed to simulate feed intake and digestion are R_i and the *fraction of feed digested in the time step* (R_d). The value of R_i can be calculated by integrating feed intake over the *time step* (Δt, h):

$$R_i = \frac{1}{v_f} \cdot \int_0^{\Delta t} r_i \cdot \left(V_{max} - V_i(t)\right) \cdot dt \tag{34}$$

Substituting the previous result and solving the integral, R_i can be calculated as:

$$R_i = \frac{1}{v_f} \cdot \frac{r_i}{r_i + r_d} \cdot \left(r_d \cdot V_{max} \cdot \Delta t + \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot \left(1 - e^{-(r_i + r_d)\cdot \Delta t}\right) \right) \tag{35}$$

Variable R_d can also be calculated in a similar way:

$$R_d = \frac{1}{V_i(0) + R_i \cdot v_f} \cdot \int_0^{\Delta t} r_d \cdot V_i(t) \cdot dt \tag{36}$$

which, when solved, yields:

$$R_d = \frac{1}{V_i + R_i \cdot v_f} \cdot \frac{r_d}{r_i + r_d} \cdot \left(r_i \cdot V_{max} \cdot \Delta t - \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot \left(1 - e^{-(r_i + r_d)\cdot \Delta t}\right) \right) \tag{37}$$

Actual feed consumption may be limited by the feeding program. If the calculated value of R_i is larger than R_a, R_i is reduced to the value of R_a. For simplicity, the value of R_d (calculated before the reduction in R_i) is assumed to remain the same. When feed intake is recorded, the vector of ingested nutrients is updated:

$$\mathbf{N_i} = \mathbf{N_i} + R_i \cdot \mathbf{N_f} \tag{38}$$

This is done before digestion is recorded, so that the ingested feed can be digested in the same time step.

Feed Digestion

The digestion process consists of transforming the ingested nutrients into digested nutrients. This involves enzyme secretion, peristalsis, absorption, fiber fermentation by intestinal flora, and other complex activities. In mathematical terms, the process is essentially a transformation of a fraction of $\mathbf{N}_I$ into a vector of digested nutrients, which is added to $\mathbf{N}_p$:

$$\mathbf{N}_p = \mathbf{N}_p + R_d \cdot \mathbf{D}_N \cdot \mathbf{N}_I \tag{39}$$

where $\mathbf{D}_N$ is a *nutrient digestion matrix*, used to represent the digestion process, which transforms a vector of ingested nutrients into a vector of digested nutrients.

With the current definitions of $\mathbf{N}_I$ and $\mathbf{N}_p$, $\mathbf{D}_N$ is defined as:

$$\mathbf{D}_N = \begin{bmatrix} d_{P\to P} & d_{L\to P} & d_{G\to P} & d_{F\to P} & 0 \\ d_{P\to L} & d_{L\to L} & d_{G\to L} & d_{F\to L} & 0 \\ d_{P\to G} & d_{L\to G} & d_{G\to G} & d_{F\to G} & 0 \end{bmatrix} \tag{40}$$

where $d_{i\to j}$ is the net change in the amount of nutrient j in the nutrient pool per unit of nutrient i ingested, i.e. the amount of absorbed j minus the amount of j consumed in the digestion process. For the current vector definitions, P, L, G and F represent protein, fat, carbohydrates and fiber, respectively. For simplicity, feed volume is assumed to have no effect on digestion, although this can easily be changed.

Negative values are possible in this matrix, and they represent nutrient consumption in the digestion process. For example, a negative value for $d_{G\to P}$ quantifies the enzyme secretion for carbohydrate digestion. Eventual negative values in $\mathbf{N}_p$ (which can occur if a nutrient is not ingested, but is required for digestion of other nutrients) will be resolved when the maintenance requirements are accounted for.

Currently, the digestible portion of fiber is assumed to be fermented and absorbed as soluble carbohydrates (based on the energy content, the volatile fatty acids formed are closer to carbohydrates than they are to fat), while all other nutrients are absorbed as themselves. It is also assumed that some amount of protein, fat and carbohydrates is secreted into the digestive tract during digestion. This includes enzymes, emulsifiers, mucus and dead cells from the walls of the digestive tract. There is also energy use and heat generation in the process of forming the secreted substances. Additional heat may be produced by the chemical reactions inside the digestive tract, including the fermentation of nutrients by the intestinal flora.

The *net energy requirement for digestion* (E_{Nd}, kJ) depends on the amount and type of feed digested:

$$E_{Nd} = R_d \cdot \mathbf{D}'_E \cdot \mathbf{N}_i \tag{41}$$

Vector $\mathbf{D'_E}$ is a digestion energy requirement vector, defined as:

$$\mathbf{D}'_E = \begin{bmatrix} d_{EP} & d_{EL} & d_{EG} & d_{EF} & 0 \end{bmatrix} \tag{42}$$

where d_{EP}, d_{EL}, d_{EG} and d_{EF} are the amounts of *net energy required to digest and absorb protein, fat, carbohydrates and fiber* (kJ / g). The energy contained in the nutrients consumed in the digestion process (already accounted for in $\mathbf{D_N}$) should not be included in these values. The energy in E_{Nd} will be added to the maintenance energy requirements further on.

The amount of heat produced in the digestion process must be added to total heat production:

$$Q_P = Q_P + R_d \cdot \mathbf{D}'_Q \cdot \mathbf{N}_i \tag{43}$$

Vector $\mathbf{D'_Q}$ is a *digestion heat vector*, defined as:

$$\mathbf{D'_Q} = \begin{bmatrix} d_{QP} & d_{QL} & d_{QG} & d_{QF} & 0 \end{bmatrix} \tag{44}$$

where d_{QP}, d_{QL}, d_{QG} and d_{QF} are the amounts of *heat generated in the process of digesting and absorbing protein, fat, carbohydrates and fiber* (kJ / g). These amounts are in addition to the heat produced by the energy usage expressed in $\mathbf{D'_E}$, so $\mathbf{D'_Q}$ should not include $\mathbf{D'_E}$.

The last step in the feed intake and digestion calculations is to determine the new values of $\mathbf{N_i}$ after digestion:

$$\mathbf{N_i} = \left(1 - R_d\right) \cdot \mathbf{N_i} \tag{45}$$

Maintenance and Tissue Catabolism

The maintenance requirements are functions of body weight, carcass composition and body temperature. The model assumes that the basal maintenance requirements are proportional to the equivalent metabolic weight ($W_e^{0.75}$). The *energy requirement for maintenance* (E_{mr}, kJ / h · kg $^{0.75}$) increases if the animal is in a hot environment. This is due to the increase in evaporative heat loss, respiratory rate and speed of chemical reactions as body temperature rises. The model assumes a linear increase in basal metabolism in response to temperature, above the evaporative critical temperature:

$$\begin{cases} E_{mr} = E'_{mr} & , T_b \le T_e \\ E_{mr} = E'_{mr} + k_r \cdot \left(T_b - T_e\right) & , T_e < T_b \end{cases} \tag{46}$$

where E'_{mr} is the thermoneutral value for E_{mr} and k_r is the *increase in maintenance energy requirement due to heat stress* (kJ / °C · h · kg $^{0.75}$).

The maintenance requirements for each nutrient in the pool are defined in the *vector of nutrient requirements for maintenance* ($\mathbf{N_{mr}}$):

$$\mathbf{N_{mr}} = \begin{bmatrix} P_{mr} \\ L_{mr} \\ G_{mr} \end{bmatrix} \tag{47}$$

where P_{mr}, L_{mr} and G_{mr} are the protein, fat and carbohydrate requirements for maintenance ($g / h \cdot kg^{0.75}$).

If tissue catabolism occurs, the amount of nutrients made available is defined in the *vector of lean tissue catabolism* ($\mathbf{N_{lc}}$) and the *vector of fat tissue catabolism* ($\mathbf{N_{fc}}$):

$$\mathbf{N_{lc}} = \begin{bmatrix} P_{lc} \\ L_{lc} \\ G_{lc} \end{bmatrix} \tag{48}$$

$$\mathbf{N_{fc}} = \begin{bmatrix} P_{fc} \\ L_{fc} \\ G_{fc} \end{bmatrix} \tag{49}$$

where P_{lc}, L_{lc} and G_{lc} are the protein, fat and carbohydrate made available by catabolism of lean tissue (g / kg) and P_{fc}, L_{fc} and G_{fc} are the protein, fat and carbohydrate made available by catabolism of fat tissue (g / kg). Each of these terms should be smaller than the corresponding term in $\mathbf{N_{lt}}$ and $\mathbf{N_{ft}}$, to account for inefficiencies in the conversion process.

Two approaches are possible when determining which tissues are catabolized to supply energy. One is to assume body mass is reduced evenly in all tissues, while the other is to assume energy comes from excess fat, unless there is none available, in which case lean tissue is catabolized. A third approach was used in this model, which is to assume that part of the energy comes from excess fat, while the remainder comes from a

uniform reduction in body weight, including both fat and lean tissues. The extent to which excess fat tissue supplies energy is determined by the *fraction of energy catabolized from excess fat* (k_E).

Maintenance Nutrient Requirements

The first step in calculating the heat and nutrient balance in the current time step is to add the heat production from decomposition of the nutrients required for maintenance to Q_P and subtract these nutrients from the nutrient pool:

$$Q_P = Q_P + \mathbf{E}'_{MN} \cdot \mathbf{N}_{mr} \cdot \Delta t \cdot W_e^{0.75} \tag{50}$$

$$\mathbf{N}_p = \mathbf{N}_p - \mathbf{N}_{mr} \cdot \Delta t \cdot W_e^{0.75} \tag{51}$$

At this point, if any of the terms in the nutrient pool is negative, it is necessary to catabolize lean tissue in order to restore the deficient nutrients. The amount of *lean tissue catabolized for nutrients* (W_{lcN}, kg) will be determined by the most limiting nutrient. For the current definition of $\mathbf{N}_p$:

$$W_{lcN} = \max\left(\frac{-N_{p[i]}}{N_{lc[i]}}\right) = \max\left(\frac{-P_p}{P_{lc}}, \frac{-L_p}{L_{lc}}, \frac{-G_p}{G_{lc}}\right) \tag{52}$$

The heat production from lean tissue catabolism is added to Q_P, the nutrients made available are added to the nutrient pool and W_{lcN} is subtracted from W_{lt}:

$$Q_P = Q_P + W_{lcN} \cdot \mathbf{E}'_{MN} \cdot \left(\mathbf{N}_{lt} - \mathbf{N}_{lc}\right) \tag{53}$$

$$\mathbf{N}_p = \mathbf{N}_p + W_{lcN} \cdot \mathbf{N}_{lc} \tag{54}$$

$$W_{lt} = W_{lt} - W_{lcN} \tag{55}$$

Maintenance Energy Requirements

The next step is to account for the energy maintenance requirements. The *net energy requirement for maintenance* (E_{Nm}, kJ) is calculated, subtracting the energy in $\mathbf{N_{mr}}$ from the energy requirement and adding the energy required for digestion (calculated earlier):

$$E_{Nm} = \left(E_{mr} - \mathbf{E'_{NN}} \cdot \mathbf{N_{mr}}\right) \cdot \Delta t \cdot W_e^{0.75} + E_{Nd} \tag{56}$$

This value is compared with the *net energy in the nutrient pool* (E_{Np}, kJ):

$$E_{Np} = \mathbf{E'_{NN}} \cdot \mathbf{N_p} \tag{57}$$

If $E_{Nm} \leq E_{Np}$, the nutrients in the pool are sufficient to provide the energy needed for maintenance. Heat generation from maintenance energy usage is added to Q_P, and the nutrients consumed are subtracted from the nutrient pool:

$$Q_P = Q_P + \mathbf{E'_{MN}} \cdot \mathbf{N_p} \cdot \frac{E_{Nm}}{E_{Np}} \tag{58}$$

$$\mathbf{N_p} = \mathbf{N_p} - \mathbf{N_p} \cdot \frac{E_{Nm}}{E_{Np}} \tag{59}$$

If $E_{Nm} > E_{Np}$, part of the energy required for maintenance must come from tissue catabolism. The amount of *fat tissue catabolized for energy* (W_{fcE}, kg) will depend on k_E, limited by the total amount of excess fat tissue:

$$W_{fcE} = \max\left(\frac{\left(E_{Nm} - E_{Np}\right) \cdot k_E}{\mathbf{E'_{NN}} \cdot \mathbf{N_{fc}}}, W_{ft} - k_{F:L} \cdot W_t\right) \tag{60}$$

The remainder of the energy deficit is supplied by catabolizing the body tissue evenly. The amount of *lean tissue catabolized for energy* (W_{lcE}, kg) and the additional W_{fcE} are proportional to the mass of each tissue:

$$W_{lcE} = W_{lt} \cdot \frac{E_{Nm} - E_{Np} - W_{fcE} \cdot \mathbf{E'_{NN}} \cdot \mathbf{N_{fc}}}{\mathbf{E'_{NN}} \cdot \left(W_{lt} \cdot \mathbf{N_{lc}} + \left(W_{ft} - W_{fcE}\right) \cdot \mathbf{N_{fc}}\right)} \tag{61}$$

$$W_{fcE} = W_{fcE} + \left(W_{ft} - W_{fcE}\right) \cdot \frac{E_{Nm} - E_{Np} - W_{fcE} \cdot \mathbf{E'_{NN}} \cdot \mathbf{N_{fc}}}{\mathbf{E'_{NN}} \cdot \left(W_{lt} \cdot \mathbf{N_{lc}} + \left(W_{ft} - W_{fcE}\right) \cdot \mathbf{N_{fc}}\right)} \tag{62}$$

Heat generation from maintenance energy usage is added to Q_P, the nutrient pool reserves are set to zero and the amounts of catabolized tissue are subtracted from the tissue mass:

$$Q_P = Q_P + \mathbf{E'_{MN}} \cdot \left(\mathbf{N_p} + W_{lcE} \cdot \mathbf{N_{lt}} + W_{fcE} \cdot \mathbf{N_{ft}}\right) \tag{63}$$

$$\mathbf{N_p} = \mathbf{0} \tag{64}$$

$$W_{lt} = W_{lt} - W_{lcE} \tag{65}$$

$$W_{ft} = W_{ft} - W_{fcE} \tag{66}$$

Additional Heat Requirements

If the animal is in a cold environment, some additional heat must be generated to maintain body temperature, which results in an increased energy requirement. The amount of heat necessary to bring body temperature back up will depend on body size and on the *specific heat of the animal's body* (c_{pb}, kJ / kg · °C). The value of c_{pb} depends on tissue composition, and can be calculated based on the *specific heat of lean tissue* (c_{pl}, kJ / kg · °C), the *specific heat of fat tissue* (c_{pf}, kJ / kg · °C) and the *specific heat of water* (c_{pw}, kJ / kg · °C):

$$c_{pb} = \frac{W_{lt} \cdot c_{pl} + W_{ft} \cdot c_{pf} + W_{lt} \cdot k_{WT} \cdot c_{pw}}{W_{lt} \cdot (1 + k_{WT}) + W_{ft}} \tag{67}$$

If $T_b < T_c$, the model calculates the *additional metabolizable energy for heat production* (E_{Mq}, kJ), which must be converted to heat to bring body temperature back up. In case of severe cold stress, E_{Mq} is fixed at its maximum value:

$$\begin{cases} E_{Mq} = \min\left((T_c - T_b) \cdot c_{pb} \cdot W_b, q_{max} \cdot W_e^{0.75} \cdot \Delta t \cdot 3.6 \frac{kJ}{W \cdot h}\right) & , T_b < T_c \\ E_{Mq} = 0 & , T_c \le T_b \end{cases} \tag{68}$$

where q_{max} is the maximum rate of additional heat production (W / $kg^{0.75}$).

The remaining calculations are similar to the ones carried out for the maintenance energy requirements, except that metabolizable energy is being used instead of net energy. First, E_{Mq} is compared with the *metabolizable energy in the nutrient pool* (E_{Mp}, kJ):

$$E_{Mp} = \mathbf{E'_{MN}} \cdot \mathbf{N_p} \tag{69}$$

If $E_{Mq} \le E_{Mp}$, the nutrients in the pool are sufficient to provide the necessary additional heat. Heat generation is added to Q_P, and the nutrients consumed are subtracted from the nutrient pool:

$$Q_P = Q_P + E_{Mq} \tag{70}$$

$$\mathbf{N_p} = \mathbf{N_p} - \mathbf{N_p} \cdot \frac{E_{Mq}}{E_{Mp}} \tag{71}$$

If $E_{Mq} > E_{Mp}$, part of the additional heat required must come from tissue catabolism. The amount of *fat tissue catabolized for heat* (W_{fcQ}, kg) will depend on k_E, limited by the total amount of excess fat tissue:

$$W_{fcQ} = \max\left(\frac{\left(E_{Mq} - E_{Mp}\right)\cdot k_E}{\mathbf{E'_{MN}}\cdot\mathbf{N_{fc}}}, W_{ft} - k_{F:L}\cdot W_{lt}\right) \tag{72}$$

The remainder of the energy deficit is supplied by catabolizing the body tissue evenly. The amount of *lean tissue catabolized for heat* (W_{lcQ}, kg) and the additional W_{fcQ} are proportional to the mass of each tissue:

$$W_{lcQ} = W_{lt}\cdot\frac{E_{Mq} - E_{Mp} - W_{fcQ}\cdot\mathbf{E'_{MN}}\cdot\mathbf{N_{fc}}}{\mathbf{E'_{MN}}\cdot\left(W_{lt}\cdot\mathbf{N_{lc}} + \left(W_{ft} - W_{fcQ}\right)\cdot\mathbf{N_{fc}}\right)} \tag{73}$$

$$W_{fcQ} = W_{fcQ} + \left(W_{ft} - W_{fcQ}\right)\cdot\frac{E_{Mq} - E_{Mp} - W_{fcQ}\cdot\mathbf{E'_{MN}}\cdot\mathbf{N_{fc}}}{\mathbf{E'_{MN}}\cdot\left(W_{lt}\cdot\mathbf{N_{lc}} + \left(W_{ft} - W_{fcQ}\right)\cdot\mathbf{N_{fc}}\right)} \tag{74}$$

Heat generation is added to Q_P, the nutrient pool reserves are set to zero and the amounts of catabolized tissue are subtracted from the tissue mass:

$$Q_P = Q_P + E_{Mq} \tag{75}$$

$$\mathbf{N_P} = \mathbf{0} \tag{76}$$

$$W_{lt} = W_{lt} - W_{lcQ} \tag{77}$$

$$W_{ft} = W_{ft} - W_{fcQ} \tag{78}$$

Tissue Deposition

After the maintenance requirements are met, deposition of lean and fat tissue may occur. The *maximum rates of lean and fat tissue deposition* (r_{ld} and r_{fd}, kg / h) are determined by the genetic potential of the animal as a function of the *age of the animal* (τ, days) and its current tissue composition:

$$r_{ld} = \frac{\phi_{ld}(\tau, W_{lt}, W_{ft})}{24 \frac{h}{day}} \tag{79}$$

$$r_{fd} = \frac{\phi_{fd}(\tau, W_{lt}, W_{ft})}{24 \frac{h}{day}} \tag{80}$$

where ϕ_{ld} and ϕ_{fd} are the growth rate functions for lean and fat tissues (kg / day).

The *energy requirements for lean tissue deposition* (E_{ld}, kJ / kg) and the *energy requirements for fat tissue deposition* (E_{fd}, kJ / kg) express the amount of net energy required per unit of tissue formed. The nutrients required per unit of lean tissue formed are defined in the *vector of nutrient requirements for lean tissue deposition* ($\mathbf{N_{ld}}$):

$$\mathbf{N_{ld}} = \begin{bmatrix} P_{ld} \\ L_{ld} \\ G_{ld} \end{bmatrix} \tag{81}$$

where P_{ld}, L_{ld} and G_{ld} are the *protein, fat and carbohydrate requirements for lean tissue deposition* (g / kg). Each of these terms is usually larger than the corresponding term in $\mathbf{N_{lt}}$. The model assumes that there is no specific nutrient requirement for excess fat tissue deposition other than the energy requirement.

Lean and Essential Fat Tissue Deposition

The amount of lean tissue deposited in the time step depends on the amount of nutrients remaining in the nutrient pool after the maintenance requirements are satisfied. The *lean tissue deposition* (W_{ld}, kg) in the time step is limited by the animal's genetic potential (given by r_{ld}), by the total net energy in the nutrient pool compared to the energy requirements, and by the amount of each nutrient in the pool compared to the nutrient

requirements. The energy required for deposition of essential fat must be added to the energy requirements:

$$W_{ld} = \min\left(r_{ld} \cdot \Delta t, \frac{\mathbf{E'_{NN}} \cdot \mathbf{N_p}}{E_{ld} + k_{F:L} \cdot E_{fd}}, \frac{N_{p[i]}}{N_{ld[i]}}\right) = \min\left(r_{ld} \cdot \Delta t, \frac{\mathbf{E'_{NN}} \cdot \mathbf{N_p}}{E_{ld} + k_{F:L} \cdot E_{fd}}, \frac{P_p}{P_{ld}}, \frac{L_p}{L_{ld}}, \frac{G_p}{G_{ld}}\right) \quad (82)$$

After W_{ld} is determined, nutrient consumption and heat production must be calculated. This is done in two steps, considering first the nutrient requirements and then the energy requirements. In the first step, the nutrients required for lean tissue deposition are removed from the nutrient pool:

$$\mathbf{N_p} = \mathbf{N_p} - W_{ld} \cdot \mathbf{N_{ld}} \quad (83)$$

In the second step, the nutrient consumption to meet the net energy requirements for lean and associated fat tissue deposition is calculated. The energy from these nutrients is added to the total heat production and the nutrients are removed from the nutrient pool:

$$Q_P = Q_P + W_{ld} \cdot \left(\left(E_{ld} + k_{F:L} \cdot E_{fd}\right) \cdot \frac{\mathbf{E'_{MN}} \cdot \mathbf{N_p}}{\mathbf{E'_{NN}} \cdot \mathbf{N_p}} - \mathbf{E'_{MN}} \cdot \left(\mathbf{N_{lt}} + k_{F:L} \cdot \mathbf{N_{ft}}\right)\right) \quad (84)$$

$$\mathbf{N_p} = \mathbf{N_p} - W_{ld} \cdot \left(\frac{\left(E_{ld} + k_{F:L} \cdot E_{fd}\right)}{\mathbf{E'_{NN}} \cdot \mathbf{N_p}} \cdot \mathbf{N_p} - \mathbf{N_{ld}}\right) \quad (85)$$

Finally, the lean and fat tissue formed is added to the animal's body tissue:

$$W_{lt} = W_{lt} + W_{ld} \quad (86)$$

$$W_{ft} = W_{ft} + k_{F:L} \cdot W_{ld} \quad (87)$$

Excess Fat Tissue Deposition

After lean tissue deposition, if there are still nutrients left in the nutrient pool to supply energy, additional fat is deposited. The *excess fat tissue deposition* (W_{fd}, kg) in the

time step is limited by the animal's genetic potential (given by r_{fd}) and by the total net energy in the nutrient pool compared to the energy requirement. The essential fat tissue already formed with lean tissue must be discounted from the maximum genetic potential:

$$W_{fd} = \min\left(r_{fd} \cdot \Delta t - k_{F:L} \cdot W_{ld}, \frac{\mathbf{E'_{NN}} \cdot \mathbf{N_p}}{E_{fd}} \right) \tag{88}$$

After W_{fd} is determined, the metabolizable energy from the consumed nutrients is added to total heat production, nutrient consumption is subtracted from the nutrient pool and W_{fd} is added to the animal's fat tissue:

$$Q_P = Q_P + W_{fd} \cdot \left(E_{fd} \cdot \frac{\mathbf{E'_{MN}} \cdot \mathbf{N_p}}{\mathbf{E'_{NN}} \cdot \mathbf{N_p}} - \mathbf{E'_{MN}} \cdot \mathbf{N_{ft}} \right) \tag{89}$$

$$\mathbf{N_p} = \mathbf{N_p} - W_{fd} \cdot \frac{E_{fd}}{\mathbf{E'_{NN}} \cdot \mathbf{N_p}} \cdot \mathbf{N_p} \tag{90}$$

$$W_{ft} = W_{ft} + W_{fd} \tag{91}$$

This completes the simulation procedure for the time step.

List of Symbols

c_{pb} Specific heat of the animal's body (kJ / kg · °C)

c_{pf} Specific heat of fat tissue (kJ / kg · °C)

c_{pl} Specific heat of lean tissue (kJ / kg · °C)

c_{pw} Specific heat of water (kJ / kg · °C)

$\mathbf{D'_E}$ Digestion energy requirement vector

d_{EF} Net energy required to digest and absorb fiber (kJ / g)

d_{EG} Net energy required to digest and absorb carbohydrates (kJ / g)

d_{EL} Net energy required to digest and absorb fat (kJ / g)

d_{EP} Net energy required to digest and absorb protein (kJ / g)

$d_{F\to G}$ Net change in G_p due to the digestion of one unit of ingested fiber

$d_{F\to L}$ Net change in L_p due to the digestion of one unit of ingested fiber

$d_{F\to P}$ Net change in P_p due to the digestion of one unit of ingested fiber

$d_{G\to G}$ Net change in G_p due to the digestion of one unit of ingested carbohydrates

$d_{G\to L}$ Net change in L_p due to the digestion of one unit of ingested carbohydrates

$d_{G\to P}$ Net change in P_p due to the digestion of one unit of ingested carbohydrates

$d_{L\to G}$ Net change in G_p due to the digestion of one unit of ingested fat

$d_{L\to L}$ Net change in L_p due to the digestion of one unit of ingested fat

$d_{L\to P}$ Net change in P_p due to the digestion of one unit of ingested fat

$\mathbf{D_N}$ Nutrient digestion matrix

$d_{P\to G}$ Net change in G_p due to the digestion of one unit of ingested protein

$d_{P\to L}$ Net change in L_p due to the digestion of one unit of ingested protein

$d_{P\to P}$ Net change in P_p due to the digestion of one unit of ingested protein

$\mathbf{D'_Q}$ Digestion heat vector

d_{QF} Heat generated in the process of digesting and absorbing fiber (kJ / g)

d_{QG} Heat generated in the process of digesting and absorbing carbohydrates (kJ / g)

d_{QL} Heat generated in the process of digesting and absorbing fat (kJ / g)

d_{QP} Heat generated in the process of digesting and absorbing protein (kJ / g)

E_c Energy concentration in the nutrient pool (kJ / kg)

E_{fd} Energy requirement for fat tissue deposition (kJ / kg)

E_{ld}	Energy requirement for lean tissue deposition (kJ / kg)
E_{max}	Maximum nutrient pool energy threshold (kJ / kg)
E'_{max}	Thermoneutral maximum nutrient pool energy threshold (kJ / kg)
E_{MG}	Metabolizable energy of carbohydrates (kJ / g)
E_{min}	Minimum nutrient pool energy threshold (kJ / kg)
E'_{min}	Thermoneutral minimum nutrient pool energy threshold (kJ / kg)
E_{ML}	Metabolizable energy of fat (kJ / g)
$\mathbf{E'}_{MN}$	Vector of metabolizable energy
E_{MP}	Metabolizable energy of protein (kJ / g)
E_{Mp}	Metabolizable energy in the nutrient pool (kJ)
E_{Mq}	Additional metabolizable energy for heat production (kJ)
E_{mr}	Energy requirement for maintenance ($kJ / h \cdot kg^{0.75}$)
E'_{mr}	Thermoneutral energy requirement for maintenance ($kJ / h \cdot kg^{0.75}$)
E_{Nd}	Net energy requirement for digestion (kJ)
E_{NG}	Net energy of carbohydrates (kJ / g)
E_{NL}	Net energy of fat (kJ / g)
E_{Nm}	Net energy requirement for maintenance (kJ)
$\mathbf{E'}_{NN}$	Vector of net energy
E_{NP}	Net energy of protein (kJ / g)
E_{Np}	Net energy in the nutrient pool (kJ)
F_f	Fiber in the feed (g / kg)
F_i	Fiber in the digestive tract (g)

G_f	Carbohydrates in the feed (g / kg)
G_{fc}	Carbohydrate made available by catabolism of fat tissue (g / kg)
G_{ft}	Carbohydrate content of fat tissue (g / kg)
G_i	Carbohydrates in the digestive tract (g)
G_{lc}	Carbohydrate made available by catabolism of lean tissue (g / kg)
G_{ld}	Carbohydrate requirement for lean tissue deposition (g / kg)
G_{lt}	Carbohydrate content of lean tissue (g / kg)
G_{mr}	Carbohydrate requirement for maintenance ($g / h \cdot kg^{0.75}$)
G_p	Carbohydrates in the nutrient pool (g)
k_E	Fraction of energy catabolized from excess fat
k_{EB}	Empty body weight fraction
k_F	Lean tissue equivalent of fat tissue
$k_{F:L}$	Minimum fat to lean ratio
k_r	Increase in maintenance energy requirement due to heat stress ($kJ / °C \cdot h \cdot kg^{0.75}$)
k_v	Relative capacity of the digestive tract ($cm^3 / kg^{0.75}$)
k_{WT}	Body water per unit of lean tissue
L_f	Fat in the feed (g / kg)
L_{fc}	Fat made available by catabolism of fat tissue (g / kg)
L_{ft}	Fat content of fat tissue (g / kg)
L_i	Fat in the digestive tract (g)
L_{lc}	Fat made available by catabolism of lean tissue (g / kg)
L_{ld}	Fat requirement for lean tissue deposition (g / kg)

L_{lt}	Fat content of lean tissue (g / kg)
L_{mr}	Fat requirement for maintenance (g / h · $kg^{0.75}$)
L_p	Fat in the nutrient pool (g)
$\mathbf{N_f}$	Vector of nutrients in the feed
$\mathbf{N_{fc}}$	Vector of fat tissue catabolism
$\mathbf{N_{ft}}$	Vector of fat tissue composition
$\mathbf{N_i}$	Vector of nutrients in the digestive tract
$\mathbf{N_{lc}}$	Vector of lean tissue catabolism
$\mathbf{N_{ld}}$	Vector of nutrient requirements for lean tissue deposition
$\mathbf{N_{lt}}$	Vector of lean tissue composition
$\mathbf{N_{mr}}$	Vector of nutrient requirements for maintenance
$\mathbf{N_p}$	Nutrient pool vector
P_f	Protein in the feed (g / kg)
P_{fc}	Protein made available by catabolism of fat tissue (g / kg)
P_{ft}	Protein content of fat tissue (g / kg)
P_i	Protein in the digestive tract (g)
P_{lc}	Protein made available by catabolism of lean tissue (g / kg)
P_{ld}	Protein requirement for lean tissue deposition (g / kg)
P_{lt}	Protein content of lean tissue (g / kg)
P_{mr}	Protein requirement for maintenance (g / h · $kg^{0.75}$)
P_p	Protein in the nutrient pool (g)
q_{max}	Maximum rate of additional heat production (W / $kg^{0.75}$)

Q_P	Heat production in the time step (kJ)
R_a	Available feed (kg)
R_d	Fraction of feed digested in the time step
r_d	Rate of digestion (h^{-1})
r_{fd}	Maximum rate of fat tissue deposition (kg / h)
R_i	Feed intake in the time step (kg)
r_i	Rate of feed intake (h^{-1})
r'_i	Thermoneutral rate of feed intake (h^{-1})
r_{ld}	Maximum rate of lean tissue deposition (kg / h)
T_b	Body temperature (°C)
T_c	Lower critical temperature (°C)
T_e	Evaporative critical temperature (°C)
T_h	Upper critical temperature (°C)
v_f	Specific volume of ingested feed in the digestive tract (cm^3 / kg)
V_i	Volume of feed in the digestive tract (cm^3)
V_{max}	Maximum capacity of the digestive tract (cm^3)
W_b	Body weight (kg)
W_e	Equivalent body weight (kg)
W_{fcE}	Fat tissue catabolized for energy (kg)
W_{fcQ}	Fat tissue catabolized for heat (kg)
W_{fd}	Excess fat tissue deposition (kg)
W_{ft}	Fat tissue weight (kg)

W_{lcE}	Lean tissue catabolized for energy (kg)
W_{lcN}	Lean tissue catabolized for nutrients (kg)
W_{lcQ}	Lean tissue catabolized for heat (kg)
W_{ld}	Lean tissue deposition (kg)
W_{lt}	Lean tissue weight (kg)
X_F	Feeding stimulus (Boolean)
Δt	Time step (h)
ϕ_{fd}	Growth rate function for fat tissue (kg / day)
ϕ_{ld}	Growth rate function for lean tissue (kg / day)
τ	Age of the animal (days)

CHAPTER 6
MICROENVIRONMENT MODULE

The Microenvironment module simulates the environment immediately surrounding the pig based on the external environment and the characteristics of the building. It would receive weather data from the Climate module and process it. However, the Climate module was not implemented in this study, so the Microenvironment module generates data by itself, based on some user defined parameters.

Cyclic Variables

Air temperature (T_a, °C), *temperature of the floor* (T_f, °C) and *temperature of the surrounding radiating surfaces* (T_r, °C) are assumed to follow a daily cycle somewhat similar to a sine wave. Minimum temperatures are assumed to occur sometime early in the morning, before sunrise, while maximum temperatures occur in the afternoon. *Relative humidity of ambient air* (ϕ_a) is assumed to follow a similar cycle, with the times of maximum and minimum inverted. *Natural air speed* (V_{an}, m / s) is assumed to cycle twice a day, being higher when air temperature is changing and lower when air temperature is stable. The time of each maximum and minimum and the values of the extremes are set by the user. The variables are calculated as a function of *time of day* (t_d, h). The model divides the curves of daily fluctuations into segments, each being a half sine wave drawn between one point of minimum and one maximum. The *intensity of*

solar radiation on a normal surface (I, W / m^2) is also a function of time, being zero at night and a symmetric curve around noon during the day.

Temperature

Minimum air temperature ($T_{a,min}$, °C), *minimum temperature of the surrounding radiating surfaces* ($T_{r,min}$, °C) and *minimum temperature of the floor* ($T_{f,min}$, °C) occur in the morning, at times $t_{Ta,min}$ (h), $t_{Tr,min}$ (h) and $t_{Tf,min}$ (h), respectively. *Maximum air temperature* ($T_{a,max}$, °C), *maximum temperature of the surrounding radiating surfaces* ($T_{r,max}$, °C) and *maximum temperature of the floor* ($T_{f,max}$, °C) occur in the afternoon, at times $t_{Ta,max}$ (h), $t_{Tr,max}$ (h) and $t_{Tf,max}$ (h), respectively. Actual temperatures for a certain time of day are given by:

$$\begin{cases} T_a = \dfrac{T_{a,min}+T_{a,max}}{2} - \dfrac{T_{a,min}-T_{a,max}}{2}\cdot\cos\left(\dfrac{t_d+24h-t_{Ta,max}}{t_{Ta,min}+24h-t_{Ta,max}}\cdot\pi\right) & , \quad t_d < t_{Ta,min} \\ T_a = \dfrac{T_{a,max}+T_{a,min}}{2} - \dfrac{T_{a,max}-T_{a,min}}{2}\cdot\cos\left(\dfrac{t_d-t_{Ta,min}}{t_{Ta,max}-t_{Ta,min}}\cdot\pi\right) & , \quad t_{Ta,min} \le t_d < t_{Ta,max} \\ T_a = \dfrac{T_{a,min}+T_{a,max}}{2} - \dfrac{T_{a,min}-T_{a,max}}{2}\cdot\cos\left(\dfrac{t_d-t_{Ta,max}}{t_{Ta,min}+24h-t_{Ta,max}}\cdot\pi\right) & , \quad t_{Ta,max} \le t_d \end{cases} \tag{92}$$

$$\begin{cases} T_r = \dfrac{T_{r,min}+T_{r,max}}{2} - \dfrac{T_{r,min}-T_{r,max}}{2}\cdot\cos\left(\dfrac{t_d+24h-t_{Tr,max}}{t_{Tr,min}+24h-t_{Tr,max}}\cdot\pi\right) & , \quad t_d < t_{Tr,min} \\ T_r = \dfrac{T_{r,max}+T_{r,min}}{2} - \dfrac{T_{r,max}-T_{r,min}}{2}\cdot\cos\left(\dfrac{t_d-t_{Tr,min}}{t_{Tr,max}-t_{Ta,min}}\cdot\pi\right) & , \quad t_{Tr,min} \le t_d < t_{Tr,max} \\ T_r = \dfrac{T_{r,min}+T_{r,max}}{2} - \dfrac{T_{r,min}-T_{r,max}}{2}\cdot\cos\left(\dfrac{t_d-t_{Tr,max}}{t_{Tr,min}+24h-t_{Tr,max}}\cdot\pi\right) & , \quad t_{Tr,max} \le t_d \end{cases} \tag{93}$$

$$\begin{cases} T_r = \dfrac{T_{f,min}+T_{f,max}}{2} - \dfrac{T_{f,min}-T_{f,max}}{2}\cdot\cos\left(\dfrac{t_d+24h-t_{Tf,max}}{t_{Tf,min}+24h-t_{Tf,max}}\cdot\pi\right) & , \quad t_d < t_{Tf,min} \\ T_r = \dfrac{T_{f,max}+T_{f,min}}{2} - \dfrac{T_{f,max}-T_{f,min}}{2}\cdot\cos\left(\dfrac{t_d-t_{Tf,min}}{t_{Tf,max}-t_{Tf,min}}\cdot\pi\right) & , \quad t_{Tf,min} \le t_d < t_{Tf,max} \\ T_r = \dfrac{T_{f,min}+T_{f,max}}{2} - \dfrac{T_{f,min}-T_{f,max}}{2}\cdot\cos\left(\dfrac{t_d-t_{Tf,max}}{t_{Tf,min}+24h-t_{Tf,max}}\cdot\pi\right) & , \quad t_{Tf,max} \le t_d \end{cases} \tag{94}$$

Relative Humidity

Maximum relative humidity of ambient air ($\phi_{a,max}$) occurs in the morning, at time $t_{\phi a,max}$ (h), while *minimum relative humidity of ambient air* ($\phi_{a,min}$) occurs in the afternoon, at times $t_{\phi a,min}$ (h). Actual relative humidity for a certain time of day is given by:

$$\begin{cases} \phi_a = \dfrac{\phi_{a,max}+\phi_{a,min}}{2} - \dfrac{\phi_{a,max}-\phi_{a,min}}{2}\cdot\cos\left(\dfrac{t_d+24h-t_{\phi a,min}}{t_{\phi a,max}+24h-t_{\phi a,min}}\cdot\pi\right) & , \quad t_d < t_{\phi a,max} \\ \phi_a = \dfrac{\phi_{a,min}+\phi_{a,max}}{2} - \dfrac{\phi_{a,min}-\phi_{a,max}}{2}\cdot\cos\left(\dfrac{t_d-t_{\phi a,max}}{t_{\phi a,min}-t_{\phi a,max}}\cdot\pi\right) & , \quad t_{\phi a,max} \le t_d < t_{\phi a,min} \\ \phi_a = \dfrac{\phi_{a,max}+\phi_{a,min}}{2} - \dfrac{\phi_{a,max}-\phi_{a,min}}{2}\cdot\cos\left(\dfrac{t_d-t_{\phi a,min}}{t_{\phi a,max}+24h-t_{\phi a,min}}\cdot\pi\right) & , \quad t_{\phi a,min} \le t_d \end{cases} \tag{95}$$

Air Speed

Natural air speed is assumed to cycle twice a day, being high in the morning and the evening, when air temperature is changing, and low during the day and at night, when air temperature is stable. *Minimum morning air speed* ($V_{a,min1}$, m / s) occurs early in the morning, at time $t_{Va,min1}$ (h). *Maximum daytime air speed* ($V_{a,max1}$, m / s) occurs at time $t_{Va,max1}$ (h), during the day. *Minimum afternoon air speed* ($V_{a,min2}$, m / s) occurs in the afternoon, at time $t_{Va,min2}$ (h). *Maximum nighttime air speed* ($V_{a,max2}$, m / s) occurs at time $t_{Va,max2}$ (h), during the evening. Actual natural air speed for a certain time of day is given by:

$$\begin{cases} V_{an} = \dfrac{V_{s,min1} + V_{s,max2}}{2} - \dfrac{V_{s,min1} - V_{s,max2}}{2} \cdot \cos\left(\dfrac{t_d + 24h - t_{Vs,max2}}{t_{Vs,min1} + 24h - t_{Vs,max2}} \cdot \pi \right) & , \quad t_d < t_{Vs,min1} \\ V_{an} = \dfrac{V_{s,max1} + V_{s,min1}}{2} - \dfrac{V_{s,max1} - V_{s,min1}}{2} \cdot \cos\left(\dfrac{t_d - t_{Vs,min1}}{t_{Vs,max1} - t_{Vs,min1}} \cdot \pi \right) & , \quad t_{Vs,min1} \le t_d < t_{Vs,max1} \\ V_{an} = \dfrac{V_{s,min2} + V_{s,max1}}{2} - \dfrac{V_{s,min2} - V_{s,max1}}{2} \cdot \cos\left(\dfrac{t_d - t_{Vs,max1}}{t_{Vs,min2} - t_{Vs,max1}} \cdot \pi \right) & , \quad t_{Vs,max1} \le t_d < t_{Vs,min2} \\ V_{an} = \dfrac{V_{s,max2} + V_{s,min2}}{2} - \dfrac{V_{s,max2} - V_{s,min2}}{2} \cdot \cos\left(\dfrac{t_d - t_{Vs,min2}}{t_{Vs,max2} - t_{Vs,min2}} \cdot \pi \right) & , \quad t_{Vs,min2} \le t_d < t_{Vs,max2} \\ V_{an} = \dfrac{V_{s,min1} + V_{s,max2}}{2} - \dfrac{V_{s,min1} - V_{s,max2}}{2} \cdot \cos\left(\dfrac{t_d - t_{Vs,max2}}{t_{Vs,min1} + 24h - t_{Vs,max2}} \cdot \pi \right) & , \quad t_{Vs,max2} \le t_d \end{cases} \tag{96}$$

Air speed can be increased to a higher *forced air speed* (V_{af}, m / s) by forced ventilation. If the fans are turned on at time $t_{Va,on}$ (h) and off at a later time $t_{Va,off}$ (h), *air speed* (V_a, m / s) will be the greatest of forced and natural air speeds between these times. If V_a is less than 0.15 m / s, its value is assumed to be 0.15 m / s (Bruce and Clark, 1979), to account for minimum natural convection. Otherwise, it will be equal to the natural air speed:

$$\begin{cases} V_a = \max\left(V_{an}, 0.15 \frac{m}{s}\right) & , \quad t_d \le t_{Va,on} \\ V_a = \max\left(V_{an}, V_{af}, 0.15 \frac{m}{s}\right) & , \quad t_{Va,on} < t_d \le t_{Va,off} \\ V_a = \max\left(V_{an}, 0.15 \frac{m}{s}\right) & , \quad t_{Va,off} < t_d \end{cases} \tag{97}$$

Solar Radiation

The values of *solar radiation on a surface perpendicular to the sun's rays at 6 a.m.* (I_6, W / m^2), *7 a.m.* (I_7, W / m^2), *8 a.m.* (I_8, W / m^2), *9 a.m.* (I_9, W / m^2), *10 a.m.* (I_{10}, W / m^2), *11 a.m.* (I_{11}, W / m^2) and *12 noon* (I_{12}, W / m^2) are supplied by the user, for the particular location and time of year in which the simulation is taking place. The model assumes solar radiation of zero between 7 p.m. and 5 a.m., and simulates a symmetrical curve around noon by interpolating data between the given values:

$$\begin{cases} I = 0 & , \quad t_d \le 5 \\ I = (t_d - 5) \cdot I_6 & , \quad 5 < t_d \le 6 \\ I = I_j + (t_d - j) \cdot (I_{j+1} - I_j) & , \quad j < t_d \le j+1 \quad , \quad j = 6,7,8,9,10,11 \\ I = I_{24-j} + (t_d - j) \cdot (I_{23-j} - I_{24-j}) & , \quad j < t_d \le j+1 \quad , \quad j = 12,13,14,15,16,17 \\ I = (19 - t_d) \cdot I_6 & , \quad 18 < t_d \le 19 \\ I = 0 & , \quad 19 < t_d \end{cases} \tag{98}$$

Fixed Variables

Some environmental parameters are simulated as fixed, not varying during the course of the day. The *atmospheric air pressure* (p_a, Pa) and the *temperature of ingested water* (T_w, °C) are defined as fixed values determined by the user. The same can be true for the *temperature of ingested feed* (T_i, °C). However, the user can specify that T_i is equal to T_a, therefore following its daily cycle.

If the user chooses to define any of the cyclic variables as constant, this may be done by setting the maximum equal to the minimum. In this case, the resulting daily cycle curve is a straight line.

List of Symbols

I	Intensity of solar radiation on a normal surface (W / m^2)
I_6	Intensity of solar radiation on a normal surface at 6 a.m. (W / m^2)
I_7	Intensity of solar radiation on a normal surface at 7 a.m. (W / m^2)
I_8	Intensity of solar radiation on a normal surface at 8 a.m. (W / m^2)
I_9	Intensity of solar radiation on a normal surface at 9 a.m. (W / m^2)
I_{10}	Intensity of solar radiation on a normal surface at 10 a.m. (W / m^2)

I_{11}	Intensity of solar radiation on a normal surface at 11 a.m. (W / m^2)
I_{12}	Intensity of solar radiation on a normal surface at 12 noon (W / m^2)
p_a	Atmospheric air pressure (Pa)
t_d	Time of day (h)
T_a	Air temperature (°C)
$T_{a,max}$	Maximum air temperature (°C)
$T_{a,min}$	Minimum air temperature (°C)
T_f	Temperature of the floor (°C)
$T_{f,max}$	Maximum temperature of the floor (°C)
$T_{f,min}$	Minimum temperature of the floor (°C)
T_i	Temperature of ingested feed (°C)
T_r	Temperature of the surrounding radiating surfaces (°C)
$T_{r,max}$	Maximum temperature of the surrounding radiating surfaces (°C)
$T_{r,min}$	Minimum temperature of the surrounding radiating surfaces (°C)
$t_{Ta,max}$	Time of day when $T_{a,max}$ occurs (h)
$t_{Ta,min}$	Time of day when $T_{a,min}$ occurs (h)
$t_{Tf,max}$	Time of day when $T_{a,max}$ occurs (h)
$t_{Tf,min}$	Time of day when $T_{a,min}$ occurs (h)
$t_{Tr,max}$	Time of day when $T_{a,max}$ occurs (h)
$t_{Tr,min}$	Time of day when $T_{a,min}$ occurs (h)
$t_{va,max1}$	Time of day when $v_{a,max1}$ occurs (h)
$t_{va,max2}$	Time of day when $v_{a,max2}$ occurs (h)

$t_{Va,min1}$ Time of day when $v_{a,min1}$ occurs (h)

$t_{Va,min2}$ Time of day when $v_{a,min2}$ occurs (h)

$t_{Va,off}$ Time of day when the fans are turned off (h)

$t_{Va,on}$ Time of day when the fans are turned on (h)

T_w Temperature of ingested water (°C).

$t_{\phi a,max}$ Time of day when $\phi_{a,max}$ occurs (h)

$t_{\phi a,min}$ Time of day when $\phi_{a,min}$ occurs (h)

V_a Air speed (m / s)

$V_{a,max1}$ Maximum daytime air speed (m / s)

$V_{a,max2}$ Maximum nighttime air speed (m / s)

$V_{a,min1}$ Minimum morning air speed (m / s)

$V_{a,min2}$ Minimum afternoon air speed (m / s)

V_{af} Forced air speed (m / s)

V_{an} Natural air speed (m / s)

ϕ_a Relative humidity of ambient air

$\phi_{a,max}$ Maximum relative humidity of ambient air

$\phi_{a,min}$ Minimum relative humidity of ambient air

CHAPTER 7
HEAT BALANCE MODULE

Overview

The Heat Balance module simulates the pig's heat loss (or gain) to the environment, heat balance and body temperature variations. The model assumes that the thermal effects of the environment on the animal can be completely expressed by the animal's *body temperature* (T_b, °C). Since body temperature is affected by all modes of heat transfer, it is believed to be a more accurate variable than ambient temperature to characterize the thermal environment zones. The fundamental differential equation on which the heat balance model is based is:

$$\frac{dT_b}{dt} = 3.6 \frac{kJ}{W \cdot h} \cdot \frac{q_P - q_L}{c_{pb} \cdot W_b} \qquad (99)$$

where $\frac{dT_b}{dt}$ is the *rate of change in body temperature* (°C / h), q_P and q_L are the *rates of heat production and heat loss* (W), c_{pb} is the *specific heat of the animal's body* (kJ / kg · °C) and W_b is the animal's *body weight* (kg).

Zones of Thermal Environment

The animal has a series of mechanisms to maintain homeostasis. The thermoneutral zone is defined as the range of environmental conditions under which the animal can

regulate heat loss with a minimum of effort. Changes in T_b alter metabolism and affect the level of heat production. If T_b drops below the *lower critical temperature* (T_c, °C), metabolism will increase in order to enhance heat production. If T_b rises above the *evaporative critical temperature* (T_e, °C), evaporative heat loss increases, via changes in the respiration rate, sweating and water diffusion to the skin. This physiological response to high temperature also causes an increase in metabolism. If T_b is between T_c and T_e, the animal is considered to be in the thermoneutral zone, where metabolism is at a minimum.

Within the thermoneutral zone, the animal is able change posture, behavior and peripheral blood flow in order to maintain body temperature approximately constant. This effect is represented in the model by changes in certain parameters, such as the heat transfer coefficient of peripheral tissue (which is altered by changes in blood flow to the skin) and the area of skin exposed to the environment. As T_b approaches T_c, these parameters will be changed so that heat loss is minimized. By the same token, if T_b approaches T_e, heat loss is maximized by changes in the parameters. The model assumes that for intermediate values of T_b, the parameters vary linearly with respect to body temperature, so that if a given parameter X has a value X_0 for $T_b \leq T_c$ and X_1 for $T_b \geq T_e$, the value of X for $T_c < T_b < T_e$ will be:

$$X = \frac{X_0 \cdot (T_e - T_b) + X_1 \cdot (T_b - T_c)}{T_e - T_c} \qquad (100)$$

Although by the traditional definitions the evaporative heat loss should be minimum below T_e, the model allows an exception to that rule. An increase in evaporation of water in the skin by wallowing in water or some other wet substance is normal in pigs, even if in a neutral environment. Wallowing involves no additional

metabolic effort and by itself is not an indication of heat stress. Therefore, the model assumes that wallowing activity (and, consequently, evaporative heat loss) increases with temperature within the thermoneutral zone.

By adjusting its metabolism, the animal is able to survive under a much wider range of environmental conditions than those of the thermoneutral zone. However, if extreme hot or cold conditions persist for a prolonged period of time, the animal will die. When heat losses to the environment exceed the maximum heat generation capacity of the animal, body temperature starts to drop rapidly and the animal becomes hypothermic. On the other extreme, evaporative heat loss reaches a maximum when T_b is at the *upper critical temperature* (T_h, °C), at which point the evaporation rate ceases to respond to further increases in T_b, and T_b starts to rise fast.

In the range of body temperatures between T_e and T_h, the animal changes its breathing and perspiration rates, in an attempt to increase evaporative heat loss and return body temperature to the thermoneutral level. The model parameters related to these changes vary accordingly, so that evaporative heat loss is reduced as T_b approaches T_e and maximized as T_b approaches T_h. The model assumes that if a given parameter Y has a value Y_0 for $T_b \leq T_e$ and Y_1 for $T_b \geq T_h$, the value of Y for $T_e < T_b < T_h$ will be:

$$Y = \frac{Y_0 \cdot (T_h - T_b) + Y_1 \cdot (T_b - T_e)}{T_h - T_e} \qquad (101)$$

Heat Transfer Across the Skin

Much of the heat transfer between the animal and its environment occurs through the animal's skin. The model considers that each portion of the skin surface can be

exposed to four different types of environments. It can either be in contact with another portion of skin (its own or another pig's), in contact with the floor, exposed to the air in a shaded environment, or exposed to the air under direct sunlight.

If the skin is in contact with another portion of skin, either by being folded over itself or in contact with other animals, heat transfer is assumed to be zero. Otherwise, heat is transferred by conduction to the floor, convection to the surrounding air, long-wave radiation from the surrounding walls, short-wave radiation from the sun and evaporation on the skin surface. The convention adopted is that heat loss is positive when flowing from the animal's body out to the environment, and negative otherwise.

Heat Transfer From Inner Body to Skin Surface

Heat transfer across the animal's skin depends on skin temperature. Most of the heat lost from the skin to the surrounding environment is produced in the inner core of the animal's body. This heat must first be transferred from the body core to the surface of the skin before it can be dissipated to the environment. The rate of heat transfer depends on the *heat transfer coefficient of the peripheral tissue* (C_t, $W / m^2 \cdot °C$) and on the difference between T_b and skin temperature. By vasodilatation and vasoconstriction, the animal is capable of increasing or decreasing C_t, within certain limits. C_t is minimum (C_{t0}, $W / m^2 \cdot °C$) for $T_b \leq T_c$ and maximum (C_{t1}, $W / m^2 \cdot °C$) for $T_b \geq T_e$.

Skin temperature may differ across the body's surface. The model defines four possible situations for the skin surface. Part of the skin may be folded over itself or in contact with other animals, in which case heat transfer is assumed to be zero. Another portion may be in contact with the floor, with all heat being lost by conduction. The

remainder of the skin surface is exposed to the air, and heat transfer occurs by convection, long-wave radiation and evaporation. Part of this surface may also be exposed to short-wave radiation from sunlight.

The *rate of heat transfer between body core and skin in contact with the floor* (q_{Rbf}, W / m^2) depends on the *temperature of the skin in contact with the floor* (T_{sf}, °C):

$$q_{Rbf} = C_t \cdot (T_b - T_{sf}) \tag{102}$$

The *rate of heat transfer between body core and skin exposed to a shaded environment* (q_{Rbe}, W / m^2) depends on the *temperature of the skin exposed to a shaded environment* (T_{sc}, °C):

$$q_{Rbe} = C_t \cdot (T_b - T_{se}) \tag{103}$$

The *rate of heat transfer between body core and skin exposed to sunlight* (q_{Rbs}, W / m^2) depends on the *temperature of the skin exposed to sunlight* (T_{ss}, °C):

$$q_{Rbs} = C_t \cdot (T_b - T_{ss}) \tag{104}$$

Heat Exchange With the Floor

Heat is transferred between the skin and the floor by conduction. The *rate of heat loss to the floor* (q_{Rf}, W / m^2) depends on the *temperature of the floor* (T_f) and on the *thermal conductance of the floor* (C_f, W / m^2 · °C):

$$q_{Rf} = C_f \cdot (T_{sf} - T_f) \tag{105}$$

Since q_{Rf} and q_{Rbf} must be equal, T_{sf} can be calculated as:

$$T_{sf} = \frac{C_t \cdot T_b + C_f \cdot T_f}{C_t + C_f} \tag{106}$$

Therefore, q_{Rf} is equal to:

$$q_{Rf} = \left(\frac{C_t \cdot C_f}{C_t + C_f} \right) \cdot (T_b - T_f) \tag{107}$$

Heat Exchange With a Shaded Environment

The skin exposed to a shaded environment is subjected to long-wave radiation from the surrounding walls, convection by movement of air and heat loss by evaporation of water. The *rate of heat loss to a shaded environment* (q_{Re}, W / m^2) is the sum of the heat loss from each of these three modes:

$$q_{Re} = \sigma \cdot e_l \cdot e_r \cdot \left(T_{sc}^{*4} - T_r^{*4} \right) + h_C \cdot (T_{sc} - T_a) + r_w \cdot h_L \cdot h_E \cdot (p_{wse} - p_{wa}) \tag{108}$$

where σ is the Stefan-Boltzmann constant ($5.6697 \cdot 10^{-8}$ W / m$^2 \cdot$ K^4), e_l is the emissivity of the skin for long-wave radiation, e_r is the emissivity of the surrounding radiating surfaces for long-wave radiation, T_r^* is the absolute temperature of the surrounding radiating surfaces (K), T_{sc}^* is the absolute temperature of the skin exposed to a shaded environment (K), h_C is the convective heat transfer coefficient (W / m$^2 \cdot$ °C), T_a is the air temperature (°C), r_w is the wetted fraction of the skin, h_L is the latent heat of evaporation of water (J / g), h_E is the convective vapor transfer coefficient (g / s $\cdot$ m$^2 \cdot$ Pa), p_{wse} is the saturation vapor pressure of air at T_{se} (Pa) and p_{wa} is the vapor pressure of the ambient air (Pa). Assuming $T_r^* \approx T_{sc}^* \approx (T_a + 273.15)$, q_{Re} can be rewritten as:

$$q_{Re} = 4 \cdot \sigma \cdot e_l \cdot e_r \cdot (T_a + 273.15)^3 \cdot (T_{se} - T_r) + h_C \cdot (T_{se} - T_a) + r_w \cdot h_L \cdot h_E \cdot (p_{wse} - p_{wa}) \tag{109}$$

where T_r is the temperature of the surrounding radiating surfaces (°C).

Evaporative heat loss is controlled by varying r_w. The animal can do this either by sweating or, if allowed to, by wallowing in mud or water. The model assumes that an increase in sweating only occurs in heat stress situations, when T_b is above T_e. However, the animal may wallow when body temperature is normal, increasing the potential evaporative heat loss in a thermoneutral environment. The model divides r_w into the *fraction of the skin wetted by perspiration* (r_{wp}) and the *fraction of skin wetted by wallowing* (r_{ww}). The value of r_{wp} is minimum (r_{wp0}) for $T_b \leq T_e$ and maximum (r_{wp1}) for $T_b \geq T_h$, while r_{ww} is minimum (r_{ww0}) for $T_b \leq T_c$ and maximum (r_{ww1}) for $T_b \geq T_e$. These two values are used to calculate r_w:

$$r_w = 1 - \left(1 - r_{wp}\right) \cdot \left(1 - r_{ww}\right) \tag{110}$$

The value of p_{wa} is obtained based on the *relative humidity of ambient air* (ϕ_a) and on the saturation vapor pressure of air at T_a:

$$p_{wa} = \phi_a \cdot \mathbf{p}_{ws}\left(T_a\right) \tag{111}$$

where $\mathbf{p}_{ws}$ is the *saturation vapor pressure as a function of temperature* (Pa). The values of h_C and h_E are functions $\mathbf{h}_C$ and $\mathbf{h}_E$ of *air speed* (V_a, m / s) and animal body weight:

$$h_C = \mathbf{h}_C\left(V_a, W_b\right) \tag{112}$$

$$h_E = \mathbf{h}_E\left(V_a, W_b\right) \tag{113}$$

Since q_{Re} and q_{Rbe} must be equal, T_{se} can be calculated as:

$$T_{se} = \frac{C_t \cdot T_b + 4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 \cdot T_r + h_C \cdot T_a - r_w \cdot h_L \cdot h_E \cdot \left(p_{wse} - p_{wa}\right)}{C_t + 4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 + h_C} \tag{114}$$

For simplicity, the value of p_{wse} from the previous time step is used to calculate T_{se}. The value of p_{wse} in the current time step is then calculated from the new value of T_{se}:

$$p_{wsc} = \mathbf{p}_{ws}\left(T_{sc}\right) \tag{115}$$

Using this new value of p_{wse}, q_{Re} is calculated as:

$$q_{Re} = \frac{C_t \cdot \left(4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 \cdot \left(T_b - T_r\right) + h_C \cdot \left(T_b - T_a\right) + r_w \cdot h_L \cdot h_E \cdot \left(p_{wsc} - p_{wa}\right)\right)}{C_t + 4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 + h_C} \tag{116}$$

Heat Exchange With a Sunlit Environment

The skin exposed to sunlight is subjected to all the heat transfer phenomena which affect skin exposed to a shaded environment and also to short-wave radiation from the sun. The *rate of heat loss to a sunlit environment* (q_{Rs}, W / m^2) is the sum of the heat loss from each of these four modes:

$$q_{Rs} = \sigma \cdot e_l \cdot e_r \cdot \left(T_{ss}^{*4} - T_r^{*4}\right) + h_C \cdot \left(T_{ss} - T_a\right) + r_w \cdot h_L \cdot h_E \cdot \left(p_{wss} - p_{wa}\right) - I \cdot \sin\theta \cdot e_s \tag{117}$$

where T_{ss}^* is the absolute temperature of the skin exposed to sunlight (K), p_{wss} is the saturation vapor pressure of air at T_{ss} (Pa), I is the intensity of solar radiation on a normal surface (W / m^2), θ is the average angle between the animal's skin and the sun's rays (a 90° angle would receive maximum radiation) and e_s is the emissivity of the skin for short-wave radiation. Assuming $T_r^* \approx T_{ss}^* \approx (T_a + 273.15)$, q_{Rs} can be rewritten as:

$$q_{Rs} = 4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 \cdot \left(T_{ss} - T_r\right) + h_C \cdot \left(T_{ss} - T_a\right) + r_w \cdot h_L \cdot h_E \cdot \left(p_{wss} - p_{wa}\right) - I \cdot \sin\theta \cdot e_s \tag{118}$$

Since q_{Rs} and q_{Rbs} must be equal, T_{ss} can be calculated as:

$$T_{ss} = \frac{C_t \cdot T_b + 4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 \cdot T_r + h_C \cdot T_a - r_w \cdot h_L \cdot h_E \cdot \left(p_{wss} - p_{wa}\right) + I \cdot \sin\theta \cdot e_s}{C_t + 4 \cdot \sigma \cdot e_l \cdot e_r \cdot \left(T_a + 273.15\right)^3 + h_C} \tag{119}$$

For simplicity, the value of p_{wss} from the previous time step is used to calculate T_{ss}. The value of p_{wss} in the current time step is then calculated from the new value of T_{ss}:

$$p_{wss} = p_{ws}(T_{ss}) \tag{120}$$

and q_{Rs} is equal to:

$$q_{Rs} = \frac{C_t \cdot \left(4 \cdot \sigma \cdot e_t \cdot e_f \cdot (T_s + 273.15)^3 \cdot (T_b - T_f) + h_C \cdot (T_b - T_a) + r_w \cdot h_L \cdot h_E \cdot (p_{wss} - p_{wa}) - I \cdot \sin\theta \cdot e_f\right)}{C_t + 4 \cdot \sigma \cdot e_t \cdot e_f \cdot (T_s + 273.15)^3 + h_C} \tag{121}$$

Body Surface Area

The animal's *body surface area* (A_b, m^2) is assumed to be proportional to body weight to the power 2/3, as a result of the relation between area and volume:

$$A_b = k_A \cdot W_b^{\frac{2}{3}} \tag{122}$$

where k_A is a constant *coefficient of body surface* per unit body weight $^{2/3}$ ($m^2 / kg^{2/3}$). The *ratio of body surface area in contact with animals* (k_a), where heat transfer is assumed zero, is a function of the environment. It can vary from zero to a maximum value of:

$$k_a = k_{a0} + k_{a1} \cdot \frac{2 \cdot (N-1)}{N} \tag{123}$$

where k_{a0} is the ratio of body surface area in contact with itself, k_{a1} is the ratio of body surface area in contact with another animal (when the animals are huddling) and N is the number of animals in each pen. The value of k_a is maximum for $T_b \leq T_c$ and zero for $T_b \geq T_e$.

The *ratio of body surface area in contact with the floor* (k_f) depends on the posture of the animal. The model assumes that the pigs will tend to maximize contact area with a cool floor in a hot environment, or with a warm floor in a cold environment. Otherwise, they will try to stay away from the floor. In order to determine which value of k_f will be used, q_{Rf} is compared with q_{Re}. If $q_{Rf} > q_{Re}$, the heat loss to the floor is greater than the

heat loss to the air, and k_f will be minimum (k_{f0}) for $T_b \leq T_c$ and maximum (k_{f1}) for $T_b \geq T_e$. On the other hand, if $q_{Rf} < q_{Re}$, the heat loss to the floor is smaller than the heat loss to the air, and k_f will be maximum (k_{f1}) for $T_b \leq T_c$ and minimum (k_{f0}) for $T_b \geq T_e$. The model assumes that there is enough space in the pen for all animals to be comfortably lying down without necessarily having to be in contact with each other.

The *ratio of body surface area exposed to sunlight* (k_s) depends on the environmental conditions and on the *availability of shade* (k_{sa}). A value of zero for k_{sa} means there is no shade where the animals can protect themselves from the sun, while a value of 1 means there is enough shade for all animals to be comfortably shaded, if they choose to be. The value of k_s will be maximum (k_{s1}) for $T_b \leq T_c$. For $T_b \geq T_e$, k_s is given by:

$$k_s = k_{s0} \cdot (1 - k_{sa}) \tag{124}$$

where k_{s0} is the minimum value for k_s in an unshaded environment.

Given that the sum of the four body surface ratios must equal unity, the *ratio of body surface exposed to a shaded environment* (k_e) can calculated:

$$k_e = 1 - (k_a + k_f + k_s) \tag{125}$$

Other Modes of Heat Transfer

Besides losing heat through the skin, the animal also heats and humidifies the air it breathes, and heats the feed and water it consumes.

Ventilation in the Respiratory Tract

When the animal breathes, the air flowing into the respiratory tract is heated to core temperature and saturated with water vapor. As the air is exhaled, it is partially cooled. The model assumes that the air leaves the respiratory tract at an intermediate temperature between air temperature and core temperature, saturated with water at that temperature. The *temperature of exhaled air* (T_x, °C) is calculated as:

$$T_x = T_b + k_x \cdot \left(T_a - T_b\right) \tag{126}$$

where k_x is the *coefficient of cooling of exhaled air*, a value between 0 and 1 which expresses the extent to which exhaled air is cooled back to ambient temperature before it leaves the respiratory tract.

The *enthalpy of exhaled air* (h_x, kJ / kg) is calculated for vapor saturated air at T_x. The *enthalpy of ambient air* (h_a, kJ / kg) is calculated using T_a and ϕ_a. Both enthalpies are calculated at *atmospheric air pressure* (p_a, Pa):

$$h_x = \mathbf{h}\left(T_x, 1, p_a\right) \tag{127}$$

$$h_a = \mathbf{h}\left(T_a, \phi_a, p_a\right) \tag{128}$$

where **h** is the enthalpy of air as a function of temperature, relative humidity and pressure (kJ / kg).

The animal's *breathing rate* (r_v, L / s) is small in cold or thermoneutral conditions, and increases under heat stress. The value of r_v is minimum (r_{v0}, L / s) for $T_b \le T_e$ and maximum (r_{v1}, L / s) for $T_b \ge T_h$. The breathing rate refers to the change in volume inside the lungs, so the *density of saturated air at body temperature* (ρ_b, kg / m^3) should be used to calculate the mass flow rate:

$$\rho_b = \rho(T_b, l, p_a) \tag{129}$$

where ρ is the density of air as a function of temperature, relative humidity and pressure (ρ_b, kg / m^3).

The *rate of heat exchange by ventilation* (q_v, W) is a function of r_v and of the difference between h_x and h_a:

$$q_v = r_v \cdot \rho_b \cdot (h_x - h_a) \tag{130}$$

Heating of Ingested Feed and Water

Feed and water ingested by the animal exchange heat with the body until they reach an equilibrium with body temperature. The amount of heat required to bring food and water to T_b depends on their temperature and on the amount ingested. *Feed intake in the time step* (R_i, kg) is generated in the Animal Metabolism module. Water ingestion is assumed to be proportional to the metabolic weight of the animal. The *rate of water consumption* (R_w, kg / h · $kg^{0.75}$) increases under heat stress, being minimum (R_{w0}, kg / h · $kg^{0.75}$) for $T_b \le T_e$ and maximum (R_{w1}, kg / h · $kg^{0.75}$) for $T_b \ge T_h$. The *rate of heat exchange with ingested matter* (q_i, W) can be calculated as:

$$q_i = \left(\frac{R_i}{\Delta t} \cdot c_{pi} \cdot (T_b - T_i) + R_w \cdot W_b^{0.75} \cdot c_{pw} \cdot (T_b - T_w) \right) \cdot \frac{W \cdot h}{3.6 kJ} \tag{131}$$

where Δt is the time step (h), c_{pi} is the specific heat of the ingested feed (kJ / kg · °C), T_i is the temperature of ingested feed (°C), c_{pw} is the specific heat of water (kJ / kg · °C) and T_w is the temperature of ingested water (°C).

Heat Balance and Body Temperature

The *total heat loss in the time step* (Q_L, kJ) is equal to the sum of all heat loss rates multiplied by the time step:

$$Q_L = \left(\left(q_{Rf} \cdot k_f + q_{Re} \cdot k_e + q_{Rs} \cdot k_s \right) \cdot A_b + q_v + q_l \right) \cdot \Delta t \cdot 3.6 \frac{kJ}{W \cdot h} \tag{132}$$

Finally, body temperature is updated for use in the next time step, based on the balance between *heat production in the time step* (Q_P, kJ), generated in the Animal Metabolism module, and heat loss:

$$T_b^{t+\Delta t} = T_b + \frac{Q_P - Q_L}{c_{pb} \cdot W_b} \tag{133}$$

This completes the simulation procedure for the time step.

List of Symbols

A_b Body surface area (m^2)

C_f Thermal conductance of the floor ($W / m^2 \cdot °C$)

c_{pb} Specific heat of the animal's body ($kJ / kg \cdot °C$)

c_{pi} Specific heat of the ingested feed ($kJ / kg \cdot °C$)

c_{pw} Specific heat of water ($kJ / kg \cdot °C$)

C_t Heat transfer coefficient of peripheral tissue ($W / m^2 \cdot °C$)

C_{t0} Minimum value for C_t ($W / m^2 \cdot °C$)

C_{t1} Maximum value for C_t ($W / m^2 \cdot °C$)

$\frac{dT_b}{dt}$ Rate of change in body temperature ($°C / h$)

e_l Emissivity of the skin for long-wave radiation

e_r Emissivity of the surrounding radiating surfaces for long-wave radiation

e_s Emissivity of the skin for short-wave radiation

h Enthalpy of air as a function of temperature, relative humidity and pressure (kJ / kg)

h_a Enthalpy of ambient air (kJ / kg)

h_C Convective heat transfer coefficient ($W / m^2 \cdot °C$)

h_E Convective vapor transfer coefficient ($g / s \cdot m^2 \cdot Pa$)

h_L Latent heat of evaporation of water (J / g)

h_x Enthalpy of exhaled air (kJ / kg)

I Intensity of solar radiation on a normal surface (W / m^2)

k_A Coefficient of body surface ($m^2 / kg^{2/3}$)

k_a Ratio of body surface area in contact with animals

k_{a0} Ratio of body surface area in contact with itself

k_{a1} Ratio of body surface area in contact with another animal

k_e Ratio of body surface exposed to a shaded environment

k_f Ratio of body surface area in contact with the floor

k_{f0} Minimum value for k_f

k_{f1} Maximum value for k_f

k_s Ratio of body surface area exposed to sunlight

k_{s0} Minimum value for k_s in an unshaded environment

k_{s1} Maximum value for k_s

k_{sa} Availability of shade

k_x Coefficient of cooling of exhaled air

N Number of animals in each pen

p_a Atmospheric air pressure (Pa)

p_{wa} Vapor pressure of the ambient air (Pa)

$\mathbf{p_{ws}}$ Saturation vapor pressure as a function of temperature (Pa)

p_{wse} Saturation vapor pressure of air at T_{se} (Pa)

p_{wss} Saturation vapor pressure of air at T_{ss} (Pa)

q_i Rate of heat exchange with ingested matter (W)

Q_L Total heat loss in the time step (kJ)

q_L Rate of heat loss (W)

Q_P Heat production in the time step (kJ)

q_P Rate of heat production (W)

q_{Rbe} Rate of heat transfer between body core and skin exposed to a shaded environment (W / m^2)

q_{Rbf} Rate of heat transfer between body core and skin in contact with the floor (W / m^2)

q_{Rbs} Rate of heat transfer between body core and skin exposed to sunlight (W / m^2)

q_{Re} Rate of heat loss to a shaded environment (W / m^2)

q_{Rf} Rate of heat loss to the floor (W / m^2)

q_{Rs} Rate of heat loss to a sunlit environment (W / m^2)

q_v Rate of heat exchange by ventilation (W)

R_i Feed intake in the time step (kg)

r_v	Breathing rate (L / s)
r_{v0}	Minimum value for r_v (L / s)
r_{v1}	Maximum value for r_v (L / s)
r_w	Wetted fraction of the skin
R_w	Rate of water consumption (kg / h · $kg^{0.75}$)
R_{w0}	Minimum value for R_w (kg / h · $kg^{0.75}$)
R_{w1}	Maximum value for R_w (kg / h · $kg^{0.75}$)
r_{wp}	Fraction of the skin wetted by perspiration
r_{wp0}	Minimum value for r_{wp}
r_{wp1}	Maximum value for r_{wp}
r_{ww}	Fraction of skin wetted by wallowing
r_{ww0}	Minimum value for r_{ww}
r_{ww1}	Maximum value for r_{ww}
T_a	Air temperature (°C)
T_b	Body temperature (°C)
$T_b^{t+\Delta t}$	Body temperature in the following time step (°C)
T_c	Lower critical temperature (°C)
T_e	Evaporative critical temperature (°C)
T_f	Temperature of the floor (°C)
T_h	Upper critical temperature (°C)
T_i	Temperature of ingested feed (°C)
T_r	Temperature of the surrounding radiating surfaces (°C)

T_r^* Absolute temperature of the surrounding radiating surfaces (K)

T_{se} Temperature of the skin exposed to a shaded environment (°C)

T_{se}^* Absolute temperature of the skin exposed to a shaded environment (K)

T_{sf} Temperature of the skin in contact with the floor (°C)

T_{ss} Temperature of the skin exposed to sunlight (°C)

T_{ss}^* Absolute temperature of the skin exposed to sunlight (K)

T_w Temperature of ingested water (°C).

T_x Temperature of exhaled air (°C)

V_a Air speed (m / s)

W_b Body weight (kg)

Δt Time step (h)

ϕ_a Relative humidity of ambient air

θ Average angle between the animal's skin and the sun's rays (°)

ρ Density of air as a function of temperature, relative humidity and pressure (kg / m^3)

ρ_b Density of saturated air at body temperature (kg / m^3)

σ Stefan-Boltzmann constant ($5.6697 \cdot 10^{-8} W / m^2 \cdot K^4$)

CHAPTER 8
MODEL IMPLEMENTATION

This chapter describes the details of the implementation of the model. Although the procedures described here are not part of the model itself, they are necessary to obtain workable values of the model's parameters.

A computer program called "SwineSim" was developed to implement a graphical user interface for the model outlined previously. SwineSim was written in the C++ programming language and runs on a personal computer under a popular graphical operating system. Although default values are hard-coded, most model parameters can be edited within the program. Furthermore, the simulation parameters can be saved into a file for later retrieval. When SwineSim starts, it looks for a file containing the default parameters. If it does not find this file, it uses the hard-coded values. This approach allows the default parameters to be changed by editing the default file.

The model is solved numerically, using Euler's first order method. The state of the system at any specific time is defined by the state variables. These variables change as time moves forward, and the value of each state variable is dependent on the state of the system from the previous time step. At each time step, the system uses the current values of the model's state variables to estimate the values for the following time step. The size of the time step can be set by the user. The default value was originally 0.1 h, which was the time step used in the model of Bridges *et al.* (1992a), but this was changed to 0.025 h,

as described in the system calibration section. Where possible, parts of the model were solved analytically, in order to improve the accuracy of the estimates.

The model was developed as a series of steps, the order of which is important. The simulation process follows the order in which these steps are described within each module. The Microenvironment is simulated first, followed by the Feeding Program and the Animal Metabolism modules. The Animal Metabolism module uses body temperature data simulated in the previous time step. Heat Balance is the last module to be simulated, because it depends on heat production data, generated in the Animal Metabolism module. At the very end of the time step, feed consumption (generated by the Animal Metabolism module) is registered in the Feeding Program module.

At the beginning of the simulation, the values entered by the user are stored in the model parameters and the state variables of the model are initialized. The implementation of each module is described separately. Except where noted as fixed, all parameters can be changed by the user.

Microenvironment

By default, minimum air ($T_{a,min}$), radiant ($T_{r,min}$) and floor ($T_{f,min}$) temperatures were set to 15 °C and maximum relative humidity ($\phi_{a,max}$) is set to 100 %, and these events were assumed to occur at 6 a.m. Default maximum air ($T_{a,max}$), radiant ($T_{r,max}$) and floor ($T_{f,max}$) temperatures were set to 25 °C and minimum relative humidity ($\phi_{a,min}$) was set to 70 %, and these events were assumed to occur at 3 p.m.

Minimum morning air speed ($V_{a,min1}$) was set to 0 m / s at 6 a.m., maximum daytime air speed ($V_{a,max1}$) was set to 2 m / s at 10 a.m., minimum afternoon air speed

($V_{a,min1}$) was set to 0 m / s at 3 p.m., Maximum nighttime air speed ($V_{a,max2}$) was set to 2 m / s at 8 p.m. Forced air speed (V_{af}) was set to 3 m/s and the fans were set to be turned on ($t_{Va,on}$) at 12 noon and off ($t_{Va,off}$) at 6 p.m.

The default solar radiation data was that of August 21 at 32° N latitude (ASHRAE, 1989): I_6 = 187 W/m^2, I_7 = 599 W/m^2, I_8 = 756 W/m^2, I_9 = 830 W/m^2, I_{10} = 869 W/m^2, I_{11} = 888 W/m^2 and I_{12} = 894 W/m^2. Atmospheric air pressure defaults to the standard atmosphere value of 101.325 kPa (ASHRAE, 1989). Default temperature of ingested water (T_w) is 20 °C and temperature of ingested feed (T_i) was assumed to follow air temperature.

Most of these are arbitrary values and the user should enter the real values for the situation being simulated.

Feeding Program

According to Whittemore (1983), feed wastage is rarely less than 2.5 % and can be up to 5 % or more. The feed use efficiency factor (e_R) was initialized to 0.95, which corresponds to a feed wastage of 5 %.

No diets were defined by default, so the user must specify a diet before running the simulation. The default feeding program had only one feeding phase with *ad lib.* feeding, which covers all of the animal's life. In this phase, only one action was defined, of type "level feed", to assure the animal always has feed available. If no feeding program is used, this should be adequate, and the user only needs to specify the diet used. Otherwise, the feeding program can be altered to reflect the actual feeding program used.

Heat Balance

Body temperature (T_b) is implemented as a state variable of the model. Though not true state variables, the skin temperatures at the various interfaces (T_{sf}, T_{se} and T_{ss}) are also carried from one time step to another. In order to initialize these temperatures correctly, the program first sets their value to the middle of the thermoneutral zone. Then, using an iterative procedure, the actual value for these temperatures is estimated. This procedure consists of running the simulation for the heat balance module without advancing time and comparing body temperature at the end of each step with its value at the beginning. When the two values are approximately equal, body temperature is consider to be initialized at a stable value.

Black *et al.* (1986) assumed that the pig maximizes sensible heat loss when body core temperature is 39 °C and maximizes total heat loss when core temperature is 40.5 °C. According to Bruce and Clark (1979), the physiological level for body temperature is 39 °C. The default values for the lower critical temperature (T_c), the evaporative critical temperature (T_e) and the upper critical temperature (T_h) were set based on this data. Assuming that body temperature varies within a range of 1 °C in the thermoneutral zone, T_c was set to 38.5 °C, T_e was set to 39.5 °C and T_h was set to 40.5 °C.

Air (T_a), radiant (T_r), floor (T_f), ingested feed (T_i) and water (T_w) temperatures, relative humidity of air (ϕ_a), air speed (V_a), intensity of solar radiation on a normal surface (I) and atmospheric air pressure (p_a) are supplied for each time step by the

Microenvironment module. Body weight (W_b) is supplied by the Animal Metabolism module.

When the pigs are in a cold environment, vasoconstriction reduces peripheral blood flow and heat transfer from the body core to the outer surface occurs primarily by conduction (Bruce and Clark, 1979). The heat transfer coefficient of peripheral tissue is then at its minimum (C_{t0}). This value was assumed to be related to a characteristic thickness of subcutaneous tissue, proportional to the cubic root of body weight, according to Bruce and Clark (1979):

$$C_{t0} = \frac{50 \frac{W \cdot kg^{\frac{1}{3}}}{m^2 \cdot °C}}{W_b^{\frac{1}{3}}} \tag{134}$$

The maximum heat transfer coefficient of the peripheral tissue was assumed to follow a similar relationship. Black *et al.* (1986) assumed that vasodilatation reduces tissue insulation to 25 percent of the vasoconstricted value. Therefore C_{t1} would be:

$$C_{t0} = \frac{200 \frac{W \cdot kg^{\frac{1}{3}}}{m^2 \cdot °C}}{W_b^{\frac{1}{3}}} \tag{135}$$

The coefficients relating C_{t0} and C_{t1} to body weight were implemented as parameters, whose default values were 50 and 200 $W \cdot kg^{1/3} / m^2 \cdot °C$, respectively.

Bruce and Clark (1979) assumed that heat flows laterally within the floor, from under the animal to the perimeter of contact. From this point of view, floor temperature (T_f) would be equal to air temperature (T_a), and the thermal conductance of the floor would be a function of thermal conductivity of the floor material and the distance heat has to travel within the floor. This distance would be affected by animal size, ratio of body surface area in contact with the floor and huddling.

Although this approach was used in the implementation of the model, it ignores heat exchange between the floor and the soil. Soil temperature fluctuates little during the day, and it generally differs from air temperature. The implementation of the thermal conductance of the floor should be revised. Nevertheless, using the values assumed by Bruce and Clark (1979), the thermal conductance of the floor (C_f) would be:

$$C_f = \frac{C_{f0}}{W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \tag{136}$$

where C_{f0} is a factor which has a value of 10.16 $W \cdot kg^{1/3} / m^2 \cdot °C$ for concrete slats, 3.09 $W \cdot kg^{1/3} / m^2 \cdot °C$ for wooden slats, 1.42 $W \cdot kg^{1/3} / m^2 \cdot °C$ for straw-bedded floors, and 1.78 $W \cdot kg^{1/3} / m^2 \cdot °C$ for insulated asphalt floors. Bruce and Clark (1979) assumed the effect of a metal mesh floor was null, and that the heat loss would be the same as the heat loss to air. However, in some circumstances, heat loss to a metal mesh floor may be greater than heat loss to the air, due to the high thermal conductivity of the metal, which acts as a heat dissipator. Using the data of Bruce and Clark (1979), C_{f0} for a metal mesh floor would be 5.93 $W \cdot kg^{1/3} / m^2 \cdot °C$.

The above values were set as choices for the user, or a custom value can be entered. The concrete slats option was set as default. Even though this approach was used, the value of T_f is supplied by the Microenvironment module as a parameter distinct from T_a. The value of k_f used in the calculation of C_f is that from the previous time step. For the first iteration, k_f is assumed to be the average of its maximum and minimum values, as described later in this chapter.

Bruce and Clark (1979) used a value of 0.99 for emissivity of pig skin for long-wave radiation (e_l) and 0.90 for that of common building materials (e_r). According to

Curtis (1983), e_l is equal to 0.95 and e_r is 0.95 for concrete and 0.90 for wood and brick. The default values used in the model were 0.95 for both parameters.

Assuming that the pig could be represented as a cylinder, Bruce and Clark (1979) calculated the convective heat transfer coefficient (h_C) as a function of air speed (V_a) and body weight (W_b):

$$h_C = 15.7 \frac{W \cdot kg^{0.13} \cdot s^{0.6}}{m^{2.6} \cdot °C} \cdot \frac{V_a^{0.6}}{W_b^{0.13}} \tag{137}$$

The convective vapor transfer coefficient (h_E, $g / s \cdot m^2 \cdot Pa$) can be calculated from h_C, using the Lewis relation (ASHRAE, 1989). If convection is considered to be a simple mixing process, the rate of mixing can be expressed as h_M, in $g_{air} / s \cdot m^2$, and it represents the rate at which a mass of air is removed from a given area of skin surface and transported to the surrounding environment. At the same time, an equal mass of air is transported from the environment to the surface of the skin. That mass of air must be heated and humidified to the same degree as the rest of the air in contact with the animal's skin. The rate of heat transfer to the air will be h_M multiplied by the temperature gradient times the specific heat of the air (c_{pa}, $kJ / kg \cdot °C$), and this is the same as h_C times the temperature gradient:

$$h_C \cdot \Delta T = h_M \cdot \Delta T \cdot c_{pa} \tag{138}$$

The rate of humidification is h_M multiplied by the humidity ratio gradient, and this is the same as h_E times the vapor pressure gradient:

$$h_E \cdot \Delta p_w = h_M \cdot \Delta W \tag{139}$$

Therefore, h_E can be calculated as a function of h_C:

$$h_E = \frac{h_C}{c_{pa}} \cdot \left(\frac{W_a}{p_{wa}} \right) \tag{140}$$

where (W_a / p_{wa}) is the relation between humidity ratio and vapor pressure. For saturated air at 20°C, $W_a = 0.014758$ g / g_{air}, $p_{wa} = 2338.9$ Pa (ASHRAE, 1989) and $c_{pa} = 0.2401$ cal / $g_{air} \cdot K = 1.0053$ J / $g_{air} \cdot$ °C (CRC, 1986), so h_E is:

$$h_E = 6.28 \cdot 10^{-6} \frac{g \cdot °C}{J \cdot Pa} \cdot h_C \tag{141}$$

which, for the current definition of h_C, is equivalent to:

$$h_E = 9.85 \cdot 10^{-5} \frac{g \cdot kg^{0.13}}{m^{2.8} \cdot s^{0.4} \cdot Pa} \cdot \frac{V_a^{0.6}}{W_b^{0.13}} \tag{142}$$

The latent heat of evaporation of water varies slightly with temperature. For a typical skin temperature of 32 °C (Bruce and Clark, 1979), h_L is 2425 J / g (ASHRAE, 1989), and this was the fixed value used in the model.

The saturation vapor pressure p_{ws} (Pa) for air at a given absolute temperature T^* (K) is calculated as:

$$p_{ws} = e^{-\frac{5674.5359}{T^*} + 6.3925247 - 0.009677843 \cdot T^* + 6.22115701 \cdot 10^{-7} \cdot T^{*2} + 0.2074782 \cdot 5 \cdot 10^{-9} \cdot T^{*3} - 9.484024 \cdot 10^{-13} \cdot T^{*4} + 4.1635019 \cdot \ln(T^*)} \tag{143}$$

for T^* between 173.15 K and 273.15 K and

$$p_{ws} = e^{-\frac{5800.2206}{T^*} + 1.3914993 - 0.048640239 \cdot T^* + 4.1764768 \cdot 10^{-5} \cdot T^{*2} - 1.4452093 \cdot 10^{-8} \cdot T^{*3} + 6.5459673 \cdot \ln(T^*)} \tag{144}$$

for T^* between 273.15 K and 473.15 K (ASHRAE, 1989).

According to Ingram and Mount (1975), water diffusion through the skin in a wide variety of animal species increases from a minimum of 10 to a maximum of 30 g / $m^2 \cdot$ h. For a 50 kg pig and 0.15 m / s air velocity, h_E would be equal to $19 \cdot 10^{-6}$ g / $s \cdot m^2 \cdot$ Pa. Assuming 50 % relative humidity of air and skin and environment temperatures of 32 °C

and 18 °C for the minimum and 35 °C and 30 °C for the maximum values of water diffusion, the vapor pressure gradient would be 3998 Pa and 3501 Pa, respectively. With these values, r_{wp0} and r_{wp1} would be 0.037 and 0.042, respectively, and these were the defaults used in the model. These values show that water diffusion only accounts for around 4 % of skin being wetted, and that this value does not change much as temperature rises. Beckett (1965) measured values from 15 to 44 $g / m^2 \cdot h$ on the middle of loin, which would make r_{wp1} up to 0.158. Although this is considerably higher than the default value, only one animal was used and moisture loss might have been overestimated. Close and Mount (1978) verified that at environmental temperatures between 10 and 15 °C the rate of evaporative heat loss was minimum and corresponded to 200 $kJ / kg^{0.75}$ per day, which is equivalent to a water loss of 1130 g per day for a 35 kg pig, but that included both respiration and water diffusion through the skin.

Assuming that wallowing does not occur below the thermoneutral zone, r_{ww0} would be equal to zero. If the animal were able to wet himself completely, r_{ww1} would be equal to one. However, if there is no water available, this value could be as low as zero. According to Black *et al.* (1986), even pigs kept entirely on slatted floors are able to wet some skin with water and urine. They assumed that the pig is able to wet up to 15 % of the skin by these means. Based on this information, the default value for r_{ww1} was set to 0.15, but the actual value will depend on the production system, and can be up to 1.

The average angle between the animal's skin and the sun's rays (θ) and the emissivity of the skin for short wave radiation (e_s) are defined as user inputs. The default value for θ was calculated by estimating the relation between the area of a cross section of the animal perpendicular to the sun's rays and the total skin area exposed to the sun.

Assuming that the shape of a pig resembles a cylinder, and that the animal could be considered a cylinder with its axis perpendicular to the sun, the ratio between these two areas can be considered to be the same ratio as that of the diameter of a circle to half of its circumference, that is, $2 / \pi$. The value of θ was calculated as the arc sine of $2 / \pi$, or approximately 40°. The default value used for e_s was 0.50, which is that of a white pig (Curtis, 1983).

The coefficient of body surface (k_A), which relates body weight to surface area, was given by Bruce and Clark (1979) as 0.09 $m^2 / kg^{2/3}$. The same value was used by Black *et al.* (1986) and was adopted as the default in the present model.

The ratio of body surface area in contact with animals can be calculated using a value of 0 for k_{a0} and 0.075 for k_{a1} (Bruce and Clark, 1979; Black *et al.*, 1986), and these values were used as defaults in the present model. Setting k_{a0} to zero has the effect of neglecting the change in exposed body surface by a single animal spreading out or curling up. When further research yields better estimates, these default values can be changed by the user. The number of animals in each pen (N) was initially set to 1, but this value should be changed to that used in the particular production system in study.

The ratio of body surface area in contact with the floor was assumed to vary between a minimum (k_{f0}) of 0.1 and a maximum (k_{f1}) of 0.2, by default (Bruce and Clark, 1979; Black *et al.*, 1986).

The model assumes by default that the animals are in a completely shaded environment. This was done by setting both k_{s0} and k_{s1} to zero. If the user wishes to simulate a different environment, these values should be changed. The value of k_{s1} should be greater than k_{s0} and neither should be greater than 0.5, except for the unlikely situation

in which mirrors are installed around the pig. The default availability of shade (k_{sa}) was 1, meaning enough shade is available. This value normally doesn't need to be changed, unless an unshaded outdoor environment is being simulated.

The coefficient of cooling of exhaled air (k_x) was calculated assuming that air leaves the respiratory tract saturated with water vapor. The data from Morrison *et al.* (1967) was used to determine the humidity ratio of the exhaled air and, consequently, the temperature of that air, from which k_x was derived. The values for k_x ranged from 0.40 to 0.79, so a default of 0.6 was assumed, meaning the exhaled air is at a temperature closer to the environmental temperature than to body core temperature. However, more research would be needed to determine the correct value of k_x.

Enthalpy (**h**) is a function of temperature, humidity and pressure. The enthalpy of air (kJ / kg) at ambient air pressure p_a (Pa), temperature T (°C) and relative humidity ϕ is (ASHRAE, 1989):

$$\mathbf{h} = T \cdot 1\,\tfrac{kJ}{kg \cdot °C} + \frac{0.62198 \cdot \phi \cdot \mathbf{p_{ws}}(T)}{p_a - \phi \cdot \mathbf{p_{ws}}(T)} \cdot \left(2501\,\tfrac{kJ}{kg} + 1.085\,\tfrac{kJ}{kg \cdot °C} \cdot T\right) \qquad (145)$$

According to Usry *et al.* (1992), maximum (r_{v0}) and minimum (r_{v1}) breathing rates are related to body surface area (A_b). Using the data of Morrison *et al.* (1967), the breathing rates varied between 0.117 L / s and 0.583 L / s per m^2 of body surface area. The actual values of r_{v0} and r_{v1} were calculated by multiplying these values (which may be overridden) by A_b:

$$r_{v0} = 0.117\,\tfrac{L}{s \cdot m^2} \cdot A_b \qquad (146)$$

$$r_{v1} = 0.583\,\tfrac{L}{s \cdot m^2} \cdot A_b \qquad (147)$$

Density of air (ρ) is a function of temperature, humidity and pressure. The density of air (kg / m^3) at ambient air pressure p_a (Pa), temperature T (°C) and relative humidity ϕ (ASHRAE, 1989) is:

$$\rho = \frac{p_a}{287.055 \frac{J}{kg \cdot K} \cdot (T + 273.15) \cdot \left(1 + 1.6078 \cdot \frac{0.62198 \cdot \phi \cdot p_{ws}(T)}{p_a - \phi \cdot p_{ws}(T)}\right)} \quad (148)$$

Van Ouwerkerk (1992) reports water intake of 5 kg / day for a 50 kg pig, which is the equivalent to 0.011 kg / h · kg$^{0.75}$. By default, the model assumes that the rate of water intake would vary between a minimum (R_{w0}) of 0.007 kg / h · kg$^{0.75}$ and a maximum (R_{w1}) of 0.015 kg / h · kg$^{0.75}$.

The default value for the specific heat of ingested feed (c_{pi}) was arbitrarily assumed to be 2 kJ / kg · °C, but this value may vary depending on feed composition. The value used for the specific heat of water (c_{pw}) was fixed at 4.1819 kJ / kg · °C, which is the value of c_{pw} for 20 °C (CRC, 1986). Considering that c_{pw} increases only 0.5 % if water temperature drops to 5 °C and does not decrease more than 0.1 % with a rise in temperature, it is reasonable to assume that this value is constant.

Animal Metabolism

Bridges *et al.* (1986) described a growth function which they later used in a simulation model (Bridges *et al.*, 1992b). The parameters they used for mature mass, time from conception when maximum growth occurred and kinetic order of the growth function were 37 kg, 280 days and 3.1 for protein, 75 kg, 325 days and 3.8 for fat, 6.75 kg, 280 days and 2.9 for ash and 126 kg, 280 days and 3.3 for water. Using these growth function parameters and assuming lean tissue is composed of the water and mineral

portions, the composition of lean tissue varied from 834 g of protein per kg of tissue at 20 kg to 848 g / kg at 100 kg, the rest of the tissue being minerals. Pomar *et al.* (1991a) assumed that the mass of minerals in the pig's body is equivalent to 21 % of the mass of protein. The default amount of protein (P_{lt}) fat (L_{lt}) and carbohydrates (G_{lt}) in lean tissue was assumed to be 842, 0 and 0 g / kg, and the protein (P_{ft}) fat (L_{ft}) and carbohydrates (G_{ft}) in fat tissue was assumed to be 0, 1000 and 0 g / kg. Although it may seem excessive to define an entire vector to hold only one useful value, this procedure simplifies future expansions of the model, when protein may be broken down into amino acids, minerals may be included, and different kinds of fat may be specified.

The values of minimum body fat to lean ratio ($k_{F:L}$) reported in the literature have been decreasing in the last few decades, as genetic improvement has been able to generate leaner pigs. Whittemore and Fawcett (1976) suggested that a minimum of one gram of fat would need to be deposited with each gram of protein. Whittemore (1983) states that a minimum level of 2 to 5 % fat exists in pigs' bodies, but this is usually exceeded in normal growth. Whittemore (1983) suggests fat : protein ratios for improved animals grown for meat of 0.5 for entire males, 0.7 for females and 0.8 for castrated males, although values as low as 0.4:1 had been observed in entire males of high lean genotype grandparent breeding stocks. Bridges *et al.* (1992a) assumed that protein and essential fat are grown together in a lean mass ratio of 0.4 units of fat deposited for every unit of protein. Pomar *et al.* (1991b) also assumed a 0.4:1 minimum fat : protein deposition ratio. Considering the composition of lean and fat tissue and using a minimum of 0.4 units of fat per unit of protein, the model assumed a default value of $0.4 \cdot 0.842 = 0.337$ for $k_{F:L}$.

Whittemore and Fawcett (1974) assumed that body weight gain could be calculated as 4.76 times protein deposition plus 1.19 times fat deposition. Pomar *et al.* (1991a) assumed that total body mass of water was 1.889 kg / $kg^{0.855}$ of total body protein. By using the growth function parameters described by Bridges *et al.* (1992b), the amount of water per unit of lean tissue was found to vary from 2.61 at 20 kg to 2.92 at 100 kg. The default amount of body water per unit of lean tissue (k_{WT}) assumed in the model was 2.78. The default empty body weight (k_{EB}) was assumed to be 0.95, or 95 % of total body weight (Pomar *et al.*, 1991a).

The masses of lean (W_{ll}) and fat (W_{fl}) tissue are implemented as state variables in the model. The initial value for these variables is determined by the starting weight and the initial fat : lean tissue ratio. The starting weight varies depending on the situation being simulated, and it should be set by the user. The main simulation routine keeps track of the animal's age. By default, the model starts with a 20 kg pig at 60 days of age.

Using the growth function parameters described by Bridges *et al.* (1992b), 20 kg pigs would have a fat : lean tissue ratio of 0.709. According to Whittemore (1983), most weaned pigs of 15 to 20 kg live weight have around 10 % fat and 16 % protein, which means a 20 kg pig would have about 3.2 kg of protein and 2 kg of fat. Using the current tissue composition, the initial fat : lean ratio would be 2 / (3.2 / 0.842) = 0.526. The default initial fat : lean tissue ratio used in the program was 0.7. However, carcass composition varies depending on genotype, sex, age and particular conditions of the production system in question. Therefore, the ratio should be change to reflect the particular situation being studied.

Many researchers express the maintenance energy requirements as a function of metabolic body weight (Whittemore and Fawcett, 1974; Close, 1978; Bruce and Clark, 1979). Others argue that metabolism is concentrated in muscles and internal organs and that the metabolic rate of fat tissue is very low. Therefore, they suggest that the maintenance requirements be expressed as a function of protein mass instead (Whittemore, 1983; Pomar *et al.*, 1991a). It is likely that the most correct approach would be to use some intermediate value, which is the reason for proposing a lean tissue equivalent of fat tissue (k_F). To consider the metabolic rate a function of body weight is the equivalent of setting k_F equal to 1. On the other hand, to use protein mass to express the maintenance requirements is the equivalent of setting k_F equal to zero. Since no references to this value exist in the literature, the intermediate value of 0.5 was assumed as default for k_F. Further research is needed to determine the correct value.

Body temperature (T_b) and its effect on model parameters with respect to the limiting temperatures of the thermal comfort zones (T_c, T_e and T_h) are implemented in the Heat Balance module.

The nutrient pool vector ($\mathbf{N_p}$) and the vector of ingested nutrients (N_i) are implemented as state variables in the model. For simplicity, the initial values of the elements of $\mathbf{N_i}$ were set to zero. To avoid divisions by zero, V_i is set to a very small value instead of zero. The elements in $\mathbf{N_p}$ are initially set so that the energy concentration in the nutrient pool is equal to the maximum thermoneutral threshold (E'_{max}) and the amount of nutrients is proportional to the amount of digestible nutrients in the feed.

According to McDonald *et al.* (1995), the average metabolizable energy value of protein (E_{MP}), fat (E_{ML}) and carbohydrates (E_{MG}) is 22.2 kJ / g, 39.0 kJ / g and 17.5 kJ / g,

respectively. However, protein is not completely oxidized, and the residual nitrogen is excreted in the urine as urea. Therefore, the actual value for E_{MP} should be reduced to 18.0 kJ / g (McDonald *et al.*, 1995). Bridges *et al.* (1992a) considered E_{MP} to be 23.86 kJ / g, E_{MG} to be 15.74 kJ / g and E_{ML} to be 39.21 kJ / g, and the efficiency of use of protein to be 84 %, resulting in a useful value of 20.04 kJ / g. Whittemore and Fawcett (1976) consider E_{MP} to be 23.6 - 7.2 = 16.4 kJ / g, and E_{ML} to be 39.3 kJ / g. According to Whittemore (1983), the energy in fat (E_{ML}) is 39.6 kJ / g and in the energy in protein is 23.7 kJ / g, but when protein is deaminated, 7.2 kJ / g are excreted in the urine and an additional 4.9 kJ / g are spent on urea formation. Pomar *et al.* (1991a) used the same assumptions. This would make E_{MP} equal to 16.5 kJ / g and the net energy of protein (E_{NP}) equal to 11.6 kJ / g, for maintenance. The default values used in the model for E_{MP}, E_{ML} and E_{MG} were 16.5, 39.6 and 17.5 kJ / g, respectively. The net energy of protein is lower than the metabolizable energy, due to the cost of forming urea. The net energy of carbohydrates and fat was considered to be the same as the metabolizable energy. Therefore, the default values for E_{NP}, E_{NL} and E_{NG} were 11.6, 39.6 and 17.5 kJ / g, respectively.

Feed composition (N_f), the amount of available feed (R_a) and the presence of a feeding stimulus (X_F) are supplied for each time step by the Feeding Program module.

Bridges *et al.* (1992a) considered 2302.7 kJ / L as the set point to determine whether the animals would be eating or not, comparing it to the energy density of the blood nutrient pools. Due to the peculiarities of the model, the thermoneutral minimum and maximum nutrient pool energy thresholds had to be determined by calibration of the

model. For reasons explained in the system validation chapter, the default values of E'_{min} and E'_{max} were set to 50 and 51 kJ / kg.

Usry *et al.* (1991) considered the maximum small intestine volume to be 285 $cm^3 / kg^{0.75}$, a function of metabolic weight. Although stomach and large intestine volumes are not included, this value is used as the default relative capacity of the digestive tract (k_v).

Bridges *et al.* (1992a) assumed that feed intake occurs at a rate of 1.32 $g / min \cdot kg^{0.75}$. Usry *et al.* (1991) used a dry matter eating rate of 0.89 $g / min \cdot kg^{0.75}$. Black *et al.* (1986) used a value of 0.111 $kg / day \cdot kg^{0.803}$ for maximum rate of feed intake, when limited by gut capacity. Pomar *et al.* (1991a) suggest that feed intake would be more adequately expressed in relation to protein mass than as a function of body weight. They propose $55.07 \cdot (1 - e^{-0.1192 \cdot PT})$ MJ / day of feed intake, where PT is the body protein mass, in kg.

Usry *et al.* (1991) assumed the density of feed dry matter to be 0.48 g / cm^3 and that of digestive tract liquid to be 1 g / cm^3. They cited proportions of up to 2.5 parts of water per unit of dry matter. Assuming feed with 90 % dry matter, each kg of feed would then occupy $1000 \cdot 0.9 \cdot (1 / 0.48 + 2.5 / 1) = 4125$ cm^3 in the digestive tract. Using this value for v_f and the currently defined value for k_v, if the eating rate is assumed to be $1 g / min \cdot kg^{0.75}$ when the digestive tract is empty, the value of r'_i would be 0.87 h^{-1}. This was used as the default in the present model.

Usry *et al.* (1991) used different gastric emptying rates for daytime (8 a.m. - 5 p.m.) and night (5 p.m. - 8 a.m.), and for up to 60 minutes after a meal and after 60 minutes post meal. The values used for day and night were 0.595 and 0.432 h^{-1} for up to 60

minutes after a meal and 0.127 and 0.103 h^{-1} for after 60 minutes post-meal. A default value of for digestion rate (r_d) of 0.4 h^{-1} was assumed in the model.

The digestion matrix represents the net transformations that occur during digestion, including nutrient usage. The coefficients $d_{P\rightarrow P}$, $d_{L\rightarrow L}$, $d_{G\rightarrow G}$ and $d_{F\rightarrow G}$ represent the digestibility of protein, fat, carbohydrates and fiber. Assuming an arbitrary apparent digestibility of 80 %, each of these coefficients would start out as 0.8.

The coefficients $d_{L\rightarrow P}$, $d_{G\rightarrow P}$ and $d_{F\rightarrow P}$ can be negative, representing the protein requirement for digestion of fat, carbohydrates and fiber (for enzyme secretion). The amount of nitrogen excreted in the feces of pigs fed nitrogen-free diet is proportional to dry matter intake (McDonald *et al.*, 1995). The endogenous protein losses vary, depending on the type of feed. McDonald *et al.* (1995) reports average values around 28 g of endogenous protein per kg of dry matter intake. Using this information, the default value for $d_{L\rightarrow P}$, $d_{G\rightarrow P}$ and $d_{F\rightarrow P}$ was assumed to be -0.028.

The secreted protein makes the apparent digestibility of protein less than the true digestibility. Assuming a protein level in the feed of around 20 %, 0.14 g of protein would be secreted per g of ingested protein. This means that the true digestibility of protein would be 0.80 + 0.14 = 0.94. However, 0.028 must be subtracted from this value, to account for the protein excretion for digestion of protein, such that $d_{P\rightarrow P}$ becomes 0.912.

It is reasonable to assume that the digestibility of fiber is smaller than the digestibility of the other components in the feed. Black *et al.* (1986) assumed digestibility of fiber to be 0.4. Assuming only half of the fiber in the feed is as digestible as the rest of the feed, $d_{F\rightarrow G}$ would need to be multiplied by 0.5. This would result in an apparent

digestibility of fiber of 0.4 for feed that is 80 % digestible. For simplification, the apparent digestibility is assumed to apply to the non-fibrous portion of the diet, which results in a lower overall digestibility of the feed as the level of fiber increases.

Black *et al.* (1986) assumed that 10 % of the fiber is transformed into methane and 6 % is lost as heat due to microbial fermentation. This fiber loss is not collected as feces, so the value of apparent digestibility does not need to be corrected. The effect on the digestion matrix was to multiply $d_{F\rightarrow G}$ by 0.84, which will resulted in a value of 0.336.

In the absence of better estimates, the default values for $d_{P\rightarrow L}$, $d_{G\rightarrow L}$, $d_{F\rightarrow L}$, $d_{P\rightarrow G}$ and $d_{L\rightarrow G}$ were assumed to be zero. The final appearance of the default digestion matrix was:

$$\mathbf{D}_N = \begin{bmatrix} 0.912 & -0.028 & -0.028 & -0.028 & 0 \\ 0 & 0.8 & 0 & 0 & 0 \\ 0 & 0 & 0.8 & 0.336 & 0 \end{bmatrix} \tag{149}$$

This matrix is correct only if all the above assumptions are satisfied, including 80 % digestibility of non-fibrous material. The coefficients of this matrix will vary depending on the type of feed used, and appropriate values for the type of feed used should be entered.

An arbitrary energy usage of 0.2 kJ per g of ingested matter (roughly 1 % of the feed's energy) was assumed to be used for digestion, which makes the digestion energy requirement vector ($\mathbf{D'}_E$) be:

$$\mathbf{D}'_E = \begin{bmatrix} 0.2 & 0.2 & 0.2 & 0.2 & 0 \end{bmatrix} \tag{150}$$

According to Black *et al.* (1986), 6 % of the digested fiber, which has an energy content of 18 kJ / g, is lost as heat. Considering a value of digestibility of fiber of 0.4, this would result in a value for heat production per unit of fiber ingested (d_{QF}) of 0.432 kJ / g.

Assuming no additional heat is generated for the other nutrients, the digestion heat vector ($\mathbf{D'_Q}$) would be:

$$\mathbf{D'_Q} = \begin{bmatrix} 0 & 0 & 0 & 0.432 & 0 \end{bmatrix} \tag{151}$$

The values for these vectors, although arbitrary, were used as default in the program. They should be changed as more specific research results become available.

Maintenance energy requirements are given in many different forms. In an attempt to compare them, a sample pig of 70 kg with a fat : lean ratio of 1 was used, assuming the current settings for the model parameters. This sample pig would have 13.9 kg of lean tissue, an equal amount of fat tissue, 11.7 kg of protein, and an equivalent weight of 62.7 kg. Close (1978) calculated the minimum maintenance energy requirement for growing pigs (E'_{mr}) of 0.440 MJ / day $\cdot$ $kg^{0.75}$, a value used by other researchers (Bruce and Clark, 1979; Bridges *et al.*, 1992a). This corresponds to a requirement per unit of equivalent metabolic weight of 19.9 kJ / h $\cdot$ $kg^{0.75}$. McDonald *et al.* (1995) reported a fasting metabolism for a 70 kg pig of 0.31 MJ / day $\cdot$ $kg^{0.75}$, or 14.0 kJ / h $\cdot$ $kg^{0.75}$ on an equivalent metabolic weight basis. Whittemore and Fawcett (1974) used 0.475 MJ / day $\cdot$ $kg^{0.75}$ as a basal energy maintenance requirement, which corresponds to 21.5 kJ / h $\cdot$ $kg^{0.75}$. The fact that these values are related to body weight would be the equivalent of setting k_F equal to 1. However, Whittemore (1983) suggested that the maintenance energy requirements should be expressed as a function of protein mass, and suggested the requirement value of 1.85 MJ / day $\cdot$ $kg^{0.78}$. Converting to equivalent metabolic weight, this corresponds to 23.6 kJ / h $\cdot$ $kg^{0.75}$. Pomar *et al.* (1991a) used the value of 2.3364 MJ / day $\cdot$ $kg^{0.74}$ of protein mass, which corresponds to 27.0 kJ / h $\cdot$ $kg^{0.75}$ of equivalent metabolic weight. Black *et al.* (1986) stated that the maintenance

requirements depend not only on body protein, but also on the rate of growth. Since the lean tissue equivalent of fat tissue was set to 0.5, the energy requirement per unit of equivalent metabolic weight should be somewhere between the requirement based on body weight and the requirement based on protein mass. The value of 20 kJ / h · $kg^{0.75}$ of equivalent body weight is used as the program's default.

Whittemore (1983) assumed zero increase in metabolism due to hyperthermic heat production, which was the equivalent of setting k_r equal to zero. This may be somewhat reasonable, considering that it is unlikely that panting consumes much energy (Ingram and Mount, 1975). The real value for k_r, although probably greater than zero, should be relatively small. Due to the lack of additional data, k_r was assumed to be zero, by default. Further research would be needed to determine the value of this parameter.

Bridges *et al.* (1992a) considered the daily maintenance protein requirements to be 0.15 % of the protein mass, based on the endogenous urinary nitrogen loss of 146 mg / day · $kg^{0.72}$ (Maynard *et al.*, 1979) for a 53 kg pig. Again using the same 70 kg reference pig, it was possible to calculate the maintenance protein requirement (P_{mr}) as a function of equivalent metabolic weight. The nitrogen requirement would be $146 \cdot 70^{0.72} = 3110$ mg / day, or 0.810 g / h of protein, which, when expressed per unit of equivalent metabolic weight, becomes 0.036 g / h · $kg^{0.75}$. Whittemore (1983) mentioned values of daily protein requirements for maintenance of 0.94 to 1.32 g / $kg^{0.75}$ of body weight, which correspond to 0.043 to 0.060 g / h · $kg^{0.75}$. Black *et al.* (1986) considered the daily nitrogen loss to be 0.15 g / $kg^{0.75}$, which corresponds to a basal protein requirement of 0.042 g / h · $kg^{0.75}$. Pomar *et al.* (1991a) expressed the daily maintenance protein requirements as 5.95 g / $kg^{0.74}$ of protein mass, which, for the reference pig, would

be equivalent to 0.069 g / h · $kg^{0.75}$ of equivalent weight. Since the requirement is based on equivalent body weight, which is an intermediate value between lean tissue and total body weight, the value 0.05 g / h · $kg^{0.75}$ was adopted as default for P_{mr}. By default, the values of L_{mr} and G_{mr} were assumed to be zero.

For lack of better estimates, the default amounts of nutrients formed from lean (N_{le}) and excess fat (N_{fe}) tissue catabolism were assumed to be equal to the tissue compositions, which implies no losses. In reality, some energy loss is expected to occur and these values are likely to be lower.

In their model, Black *et al.* (1986) assumed that any excess or deficiency of metabolizable energy is allocated to or taken from fat reserves, which is the equivalent of setting k_E equal to 1. Bridges *et al.* (1992a) considered that both fat and lean tissues are catabolized to supply energy, which would be the equivalent of setting k_E equal to zero. The default value of k_E was initially assumed to be 0.5, but, as described in the system validation chapter, was later changed to 0.87.

According to Ingram and Mount (1975), the specific heats of lean tissue (c_{pl}) and fat (c_{pf}) are 3.47 and 1.88 and kJ / kg · °C, and these were the default values used. The specific heat of water (c_{pw}) was fixed at 4.1819 kJ / kg · °C (CRC, 1986). The maximum additional heat production (q_{max}) was arbitrarily set to the same value as the maintenance energy requirement, which was 20 kJ / h · $kg^{0.75}$, or 5.56 W / $kg^{0.75}$.

According to Schinckel and Lange (1996), metabolizable energy intake for protein and fat deposition have been estimated at 44.1 and 52.9 kJ / g, respectively. McDonald *et al.* (1995) mentioned requirements of 42.3 and 53.5 kJ / g for protein and fat. Whittemore (1983) suggested an energy cost of 43.9 kJ / g for protein deposition, which includes

23.7 kJ / g of energy in the protein itself. Whittemore (1983) assumed that the energy cost for fat deposition was 53.5 kJ / g, which included the energy content of fat (39.6 kJ / g). Pomar *et al.* (1991a) also considered the energy cost of protein and fat deposition to be 43.9 and 53.5 kJ / g, respectively. The energy costs for protein and fat deposition used by Whittemore and Fawcett (1974) were 60.0 kJ / g and 53.5 kJ / g. The value for protein was unusually high, compared to most of the literature.

Black *et al.* (1986) used an efficiency of utilization of metabolizable energy for maintenance of 0.80, and efficiencies of 0.54 and 0.74 for protein and fat deposition for their model. Bridges *et al.* (1992a) considered the conversion efficiency of protein, fat and carbohydrates to be 0.44, 0.60 and 0.63 for protein growth and 0.50, 0.90 and 0.73 for fat growth. However, the protein inefficiencies included the deamination inefficiencies, already accounted for when calculating net energy.

Considering the current values for tissue composition, the default energy requirement for fat tissue deposition (E_{fd}) was assumed to be $53.5 \cdot 1000 = 53500$ kJ / kg. In the present work, the energy value for protein always excluded the 7.2 kJ / g that is excreted as urea when protein is deaminated. Therefore, the default energy cost of lean tissue deposition (E_{ld}) was considered to be $(43.9 - 7.2) \cdot 842 = 30901$ kJ / kg.

According to Whittemore (1983) the efficiency of protein deposition is probably between 0.85 and 1.00. A value of 0.9 was used in the present study, which, based on the tissue composition results in a default protein requirement for lean tissue deposition (P_{ld}) of 842 / 0.9 = 936 g / kg. The default protein deposition requirements of fat (L_{ld}) and carbohydrates (G_{ld}) were assumed to be zero.

Growth Curves

The potential growth rates for both lean and fat tissues are defined as functions of age, lean tissue mass and fat tissue mass. The proposed simulation model is independent of the growth curve, such that any growth curve which can be expressed in terms of these three parameters can be used in the model. Currently, two growth curves are defined, and the user can select the one that should be used. One is a linear growth curve, in which the growth rate was assumed to be constant. The other is a growth curve based on physiological age, described by Bridges *et al.* (1986) and used by Bridges *et al.* (1992a,b). Each growth curve uses a set of parameters, which can be defined by the user. In order to simulate different genotypes or sexes, one needs only to change the value of the parameters accordingly.

Linear Growth Model

The first growth rate function defined is the simplest possible function, which defines growth rate as constant. It has only one parameter for each curve (the growth rate itself). Growth is defined by the differential equations:

$$\frac{dW_{ft}}{dt} = G1k_L \tag{152}$$

$$\frac{dW_{ft}}{dt} = G1k_F \tag{153}$$

where $G1k_L$ (kg / day) and $G1k_F$ (kg / day) are the growth rates for lean and fat tissue, respectively. This may be adequate for short term simulations, but not for longer simulation times. This approach may be useful if only a few days are simulated, in order

to quickly verify the effect of changing potential growth rate on the simulation results. For a 70 kg animal, possible values are 0.204 kg / day for $G1k_L$ and 0.150 kg / day for $G1k_F$. These values were derived from the physiological age growth curve, described next, and were used as defaults by the model for this growth function.

Physiological Growth Model

The second growth rate function defined is the one described by Bridges *et al.* (1986), which is based on physiological age. Growth is defined in this model by the differential equations:

$$\frac{dW_{lt}}{dt} = G2W_{maxL} \cdot (G2a_L - 1) \cdot \frac{t^{G2a_L - 1}}{G2t_{maxL}^{G2a_L}} \cdot e^{-\frac{G2a_L - 1}{G2a_L} \cdot \frac{t^{G2a_L}}{G2t_{maxL}^{G2a_L}}} \tag{154}$$

$$\frac{dW_{ft}}{dt} = G2W_{maxF} \cdot (G2a_F - 1) \cdot \frac{t^{G2a_F - 1}}{G2t_{maxF}^{G2a_F}} \cdot e^{-\frac{G2a_F - 1}{G2a_F} \cdot \frac{t^{G2a_F}}{G2t_{maxF}^{G2a_F}}} \tag{155}$$

where $G2W_{maxL}$ (kg) and $G2W_{maxF}$ (kg) are the masses of mature lean and excess fat tissue, $G2a_L$ and $G2a_F$ are the kinetic orders of the lean and excess fat growth rate functions, $G2t_{maxL}$ (days) and $G2t_{maxF}$ (days) are the times from conception when the instantaneous maximum lean and excess fat growth rate occurs, and t (days) is the physiological age, starting from conception. Using the concepts described by Loewer *et al.* (1987), the physiological age can be calculated as a function of lean tissue mass by inverting the growth function for lean tissue:

$$t = G2t_{maxL} \cdot \left(-\ln\left(1 - \frac{W_{lt}}{G2W_{maxL}}\right) \cdot \frac{G2a_L}{G2a_L - 1} \right)^{\frac{1}{G2a_L}} \tag{156}$$

As an alternative, the chronological age may be used instead of the physiological age, in which case t is calculated by adding the gestation period to the age of the animal. By default, SwineSim uses this model of growth curves and calculates the physiological age from lean tissue mass. The default gestation period was 114 days, and it must be added to the chronological age, if it is to be used in the model.

Bridges *et al.* (1992b) used in their model a mature mass of 37 kg for protein, 75 kg for fat, 6.75 kg for ash and 126 kg for water. The time of maximum growth rate used was 280 days from conception for all tissues except fat, in which it was 325 days. The kinetic order of the growth rate function used was 3.1 for protein, 3.8 for fat, 2.9 for ash and 3.3 for water. Using these values and combining the protein and ash curves, it was possible to arrive at numerical approximations for the growth curve parameters for lean tissue. The parameters for the fat growth curve were not modified. The values obtained were 43.75 kg, 75 kg, 3.068, 3.8, 280 days and 325 days for $G2W_{maxL}$, $G2W_{maxF}$, $G2a_L$, $G2a_F$, $G2t_{maxL}$ and $G2t_{maxF}$, respectively, and these values are used as default in the program.

CHAPTER 9
EXPERT SYSTEM

After the simulation model runs, the output is analyzed by an expert system, which provides the user with additional information about the system being simulated. The objective of the expert system in this study was to show how it can be used to improve the usefulness of a simulation model and to demonstrate how the two can be interfaced. The quality of the expert system's knowledge base was not a concern, given that it can always be improved in the future. Therefore, a very simple knowledge base was used and the effort was concentrated on the interface.

Inference Engine

The expert system uses output from the simulation program to suggest actions which will improve the performance of the production system. The expert system consists of a set of parameters (the data base), a set of rules (the knowledge base) and a program (the inference engine) that selectively executes the rules, depending on the values of the parameters.

The parameters are implemented as real numbers, and the meaning of their value depends on each individual parameter. Each parameter is initialized to an invalid value at the beginning of the expert system session, in order to avoid inadvertently firing a rule. As the simulation output data base is processed, some parameters may be set to valid

values. Each time a parameter is set, the inference engine tests all the rules that depend on the parameter. If certain conditions defined in the premise of the rule are satisfied, the rule will fire.

When a rule fires, it supplies the user with some information about the production system being analyzed. This may include measurements of system efficiency, suggestions for changing the system, and any other information which may be useful. When a rule fires, it may also change the value of some other parameters, which can then cause other rules to fire. In order to simplify the implementation of the inference engine and avoid endless loops, each rule is allowed to fire only once per session.

In order to implement the expert system as described above, the inference engine was developed as two separate modules. One is the inference engine which is responsible for all the processing of information. The other is the knowledge base, in which the rules are defined, along with the parameters used by them. The inference engine maintains a list of parameters and a list of rules. When either is defined in the knowledge base, it is automatically added to the corresponding list. Each parameter contains a list of all the rules whose premises depend on the value of the parameter. When the rules are defined, they must register themselves with the appropriate parameters. The way to do this is explained further down.

Both the inference engine and the knowledge base were implemented in C++. It is important to understand the format of the knowledge base code, in order to be able to expand the expert system knowledge base in the future. Parameters are implemented as objects of a class named 'Parameter', while rules are implemented as classes derived from a base class 'Rule'.

The constructor for a parameter takes two arguments. The first is a string with the name of the parameter, and the second is the default value (the value considered invalid). For example, if a parameter called 'BodyTemperature' was to be declared with a default value of -50 °C (an obviously invalid value), the corresponding C++ code would be:

```
Parameter BodyTemperature ("BodyTemperature", -50);
```

Alternatively, if the default value of the parameter is zero, it may be omitted. The following code would define a parameter called 'Hyperthermia' with a default value of zero:

```
Parameter Hyperthermia ("Hyperthermia");
```

In this case, a convention may be adopted that the value of 'Hyperthermia' is 1 if the animal is hyperthermic, 2 if it is not, and 0 if it is not known whether the animal is hyperthermic or not.

In order to define a rule, a class must be derived from class 'Rule' and an object of the derived class must be declared. Three functions must be defined and made public in the derived class. The first is the constructor, which has the same name as the class. The other two are 'int Premise (void)' and 'void Action (void)', which are implemented in 'Rule' as pure virtual functions and must be overridden. As an example, a rule called 'Rule1' is defined and an object 'R1' of this class is declared:

```
class Rule1:Rule {
public:
     Rule1 ();
     int Premise (void);
     void Action (void);
} R1;
```

The 'Premise' function should return zero if the premise of the rule is not satisfied, and a value other than zero if the premise is satisfied and the rule is to be executed.

Supposing this rule is to be fired when body temperature rises above 40 °C, the corresponding code would be:

```
int Rule1::Premise (void) {
     return (BodyTemperature.Get() > 40);
};
```

As seen in the example, the value of a parameter is obtained using the parameter's 'Get' function with no arguments. The 'Action' function defines the actions to be taken if the rule is to be executed. Suppose, in this example, that whenever 'Rule1' fires the user should be informed that the animal is hyperthermic and the parameter 'Hyperthermia' should be set to 1. The corresponding code would be:

```
void Rule1::Action (void) {
     Ex.Log ("The animal is hyperthermic.");
     Hyperthermia.Set (1);
};
```

As shown, the value of a parameter is set using the parameter's 'Set' function, with the corresponding value as argument. Also, messages are sent to the user via the function 'Ex.Log'. 'Ex' is the object that implements the expert system class. Finally, the constructor for the rule must be defined. The constructor must set the value of the variable 'Name' and register the rule with the parameters on which it depends by using the parameter's 'UseRule' function, with 'this' as argument. In this example, the only parameter used in the rule's premise is 'BodyTemperature':

```
Rule1::Rule1 () {
     Name = "Rule1";
     BodyTemperature.UseRule (this);
};
```

The example above shows how the knowledge base should be implemented. The simulation program automatically executes the expert system every time it finishes a simulation run. When this happens, the inference engine is initialized, and a list of

parameters resulting from the simulation output is set, one at a time. This may fire rules which may produce output or set other parameters. After the inference engine finishes processing the simulation output, it is reset and the user can view the expert system results.

Knowledge Base

As mentioned before, a very simple knowledge base is implemented with the model. Only two rules are defined, related to heat and cold stress.

The cold stress rule fires when over 1 % of the simulated pig's heat production is used for additional heat to maintain body temperature stable and body temperature is below the lower critical temperature for over 1 % of the time.

When this rule is fired, it informs the user what percentage of total heat production was for additional heat, how long body temperature was below the lower critical temperature for, and what the average body temperature was during this time. It also suggests that actions should be taken to heat the environment during the cold periods. A typical output when this rule fires is:

```
3.51 % of the animal's heat production was for supplemental heat.
The environment was below the thermoneutral zone for 31.88 % of the
time.
During this time, average body temperature was 38.49°C.
The system should be changed to heat the environment during these
periods.
```

The heat stress rule fires when the tissue growth rate drops a certain amount below the animal's potential and body temperature rises for a certain period of time. The minimum growth rate reduction to fire this rule is 1 % for lean tissue and 5 % for fat tissue. If either of these conditions occur, the rule considers growth rate reduction to have

occurred. Body temperature is considered to be high if it is either above the evaporative critical temperature for over 5 % of the time or above the upper critical temperature for more than 1 % of the time.

When this rule is fired, the user is informed the reduction amount in lean and fat tissue growth, relative to the maximum potential. The amount of time that body temperature was above the lower critical temperature and the average body temperature during this period is also informed, as well as how much time body temperature was above the upper critical temperature and the average body temperature for that period. The rule also suggests that actions should be taken to cool the environment during the hot periods. A typical output when this rule fires is:

```
The animal was growing lean tissue 2.64 % below its potential and
fat tissue 3.87 % below its potential.
The environment was above the thermoneutral zone for 22.76 % of the
time.
During this time, average body temperature was 40.92°C.
15.05 % of the time the animal was hyperthermic.
During this time, average body temperature was 41.44°C.
The system should be changed to cool the environment during these
periods.
```

The knowledge base currently contain only these two simple rules. The idea was to demonstrate how the expert system could be interfaced with the simulation model, so no significant effort was made toward developing more rules. As the knowledge base is expanded, the expert system will prove to be more useful.

CHAPTER 10
SOFTWARE DESCRIPTION

The simulation model was implemented as a computer program called "SwineSim". SwineSim was developed using a graphical programming environment associated with a C++ compiler. The user interface is designed as a series of forms. The forms contain components, both visual and non-visual. Forms and components have associated properties, methods and event handlers. Two forms were designed: the Main Window form, where all the visual input and output of the program occurs, and an About Box. The About Box, on Figure 3, shows some information about the program, and is accessed by selecting Help | About in the program's main menu.

The other option of the main menu deals with the normal file operations (File | New, File | Open, File | Save, File | Save As) and exiting the program (File | Exit). SwineSim files have the default extension '.swi', and they consist of text files containing the values of the model parameters used in the simulation. After a simulation runs, two output files are generated. Both files contain the values of all the output variables used in the graphs separated by tabs, with one line per observation, in such a way that they can easily be imported into a spreadsheet. The first line of the file contains the name of the variables, also separated by tabs. The difference between the two files is in the frequency of output.

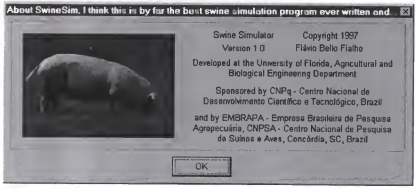


Figure 3. SwineSim About Box.

The file of general results has the same name as the input file, with extension '.swr', and it contains all the observations used in the graphs. The time interval between the observations is 0.1 h for the first two days of the simulation run, 0.2 h for the next two days, 0.4 h for the next four days, 0.8 h for the days 8 through 15, 1.6 h for days 16 through 31, 3.2 h for days 32 through 63, 6.4 h for days 64 through 127, 12.8 h for days 128 through 255 and 25.6 h for days 256 through 512. No data is recorded after day 512. Counting of time resets every time the simulation stops. This file is useful for obtaining precise information for short term runs, and doesn't become excessively large for long term simulations.

The file of daily results has the same name as the input file, with extension '.swr', and it contains only one observation every 24 hours. Counting of time only resets when the simulation resets, and not when it stops. This file is not useful for details about a particular day of the simulation, but it contains data for all the simulated days and is better for analyzing and graphing long time periods.

The main window is subdivided into tabs. Each tab shows a different type of screen interface. The Simulation tab accepts some simulation parameters, such as the length of

the time step, some initial parameters (age, weight, carcass composition and an estimate of heat production) and the finishing condition (either the length of time the simulation should run or the finishing weight). The simulation tab also displays a graph with the simulation results. The type of graph shown can be selected from a list of 15 different graphs.

The simulation tab is shown in Figure 4. The two buttons on the left side of the screen are used to control the execution of the simulation program. The first button is used to start the simulation if it has not started, stop the simulation if it is running, or continue the simulation if it has stopped. The second button is used to reset the simulation, so it will start from the initial condition. After the simulation finishes, the user may change the parameters of the model and continue to run the simulation from where it left off. Most of the parameters can be changed freely at this point, but the starting age, weight and fat : lean ratio are only reloaded when the reset button is pressed.

When the simulation is running, the second button serves to pause the simulation so that the graphs can be examined. Otherwise, the graphs are being constantly updated and may be difficult to visualize in detail. The difference between stopping and pausing the simulation is in the graphed output. If the simulation is paused and then resumed, the graph continues to display data as if nothing had happened. On the other hand, if the simulation is stopped and then continued, the starting point of the graph is reset to the time when the simulation stopped. This allows the user to simulate data for a long period of time and then continue the simulation for a couple of days more, to see the graphs for those days in more detail. Depending on the state of the system, the first button may be

labeled 'Start', 'Stop' or 'Continue', and the second button may be labeled 'Reset', 'Pause' or 'Resume'.

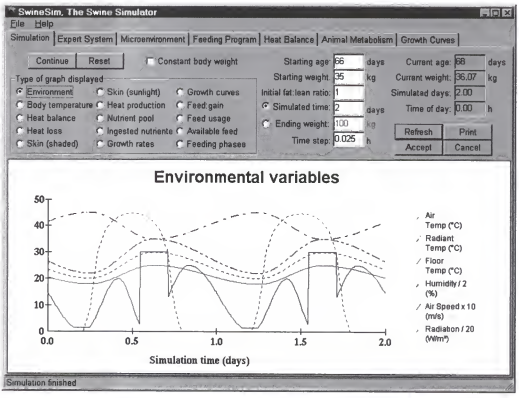


Figure 4. SwineSim simulation of environmental variables.

To the right of the buttons, there is a 'Constant body weight' option, which, when checked, resets the values of lean and fat tissue mass at the end of each time step. This is useful to examine steady state conditions when the animal has a specific body composition. Growth rates will still be displayed correctly, for that particular instant. The starting age, starting weight and initial fat : lean ratio determine the age and body composition of the animal when the simulation is reset. These values are read when the simulation starts, as long as the reset button has been pressed or a file has been loaded.

The simulation will run for a certain number of days or until the animal reaches a given weight, both of which can be specified by the user. By selecting one of the radio buttons, the user specifies which of the two conditions must be satisfied. Note that if the ending weight is selected the simulation might run forever if the animal does not gain weight. In this case, the user will have to press the 'Stop' button to manually stop the simulation. Below these options is the time step to be used in the simulation.

On the right side of the window, the program displays the current age, weight, time from the start of the simulation and current simulated time of day. These values cannot be changed by the user. In the 'Simulation' tab and in some of the other tabs the four buttons labeled 'Refresh', 'Print', 'Accept' and 'Cancel' are present. The 'Refresh' button updates the graph, while the 'Print' button prints it. 'Accept' tells the system that the currently entered values should be accepted, while 'Cancel' erases the newly edited values and replaces them with the previously accepted values. All edited values are automatically accepted when the simulation starts or continues (but not when it resumes after a pause) and when the simulation parameters are saved to a file.

The type of graph displayed on the lower portion of the screen may be changed by selecting one of the radio buttons. Following is a description of the types of graphs that can be selected. Two examples were simulated for these graphs one is a 2-day trial with feeding restriction and the other is a 100-day trial, in which feeding was restricted in the first and third phases and unrestricted in the second phase. Note that the graphs appear in color on the computer screen, which greatly facilitates visualization.

Figure 4 shows the graph of environmental variables for a two-day simulation run. Some graphs, such as this one, are designed to view the results of short-term simulations

and look confusing for longer simulation periods. The environmental variables displayed as a function of time are air temperature (°C), temperature of the surrounding radiating surfaces (°C), temperature of the floor (°C), relative humidity (%), air speed (m / s) and intensity of solar radiation on a normal surface (W / m^2). The scale on the Y-axis is for temperature. Each tick (10 units) represents 20 % relative humidity, 1 m / s and 200 W / m^2 for the other variables. These values are all simulated in the Microenvironment module.

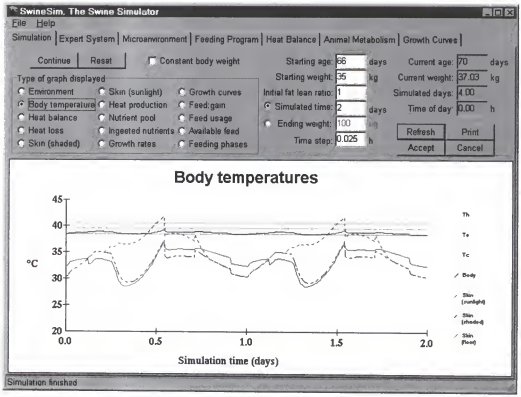


Figure 5. SwineSim simulation of body temperatures.

The graph of simulated body temperatures is shown in Figure 5. The three parallel lines at 38.5, 39.5 and 40.5 °C represent the lower (T_c), evaporative (T_e) and upper (T_h)

critical temperatures set by the user. Simulated body temperature (°C) is shown in this graph, and its value relative to the parallel lines indicates the degree of thermal comfort of the animal. In this example, the animal is generally in the lower portion of the thermoneutral zone, although body temperature reaches T_c at some points. The other three temperatures shown are the simulated temperatures of the skin exposed to sunlight, skin exposed to a shaded environment and skin in contact with the floor. As expected, the graph shows that skin temperature varies much more than body core temperature.

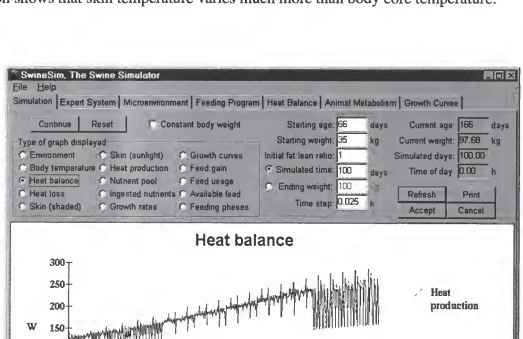


Figure 6. SwineSim simulation of heat balance.

The heat balance graph in Figure 6 contains the rates of heat production (W) and heat loss (W). These two curves should be the same, if body temperature is to remain

constant. The animal is able to compensate for an increase in heat loss by increasing heat production and vice-versa, but there is usually some amount of lag between these responses, so minor fluctuations in body temperature are normal. This graph can be used to view the oscillations in the heat balance during the course of a day, or, as seen in Figure 6, to view how the level of heat production changes in a longer period of time. In this example, the periods of greater oscillation in heat production and heat loss correspond to periods of restricted feeding, when the animal alternated between fasting heat production and heat production including tissue deposition. In the intermediate period, from 30 to 78 days, feeding was unrestricted, therefore heat production remained at the higher level. It is also possible to visualize the increase in heat production as the animal gets older.

The partitioning of heat loss is shown on the graph in Figure 7. In this graph it is possible to visualize the amount of heat (W) lost through the skin exposed to a shaded environment, through the skin exposed to sunlight, through the skin in contact with the floor, through heating and humidifying air by breathing, and through heating of ingested feed and water. It can be seen here that most of the heat is lost through the skin to a shaded environment (the animals were kept away from sunlight, in this case), followed by heat loss to the floor. If this graph is compared to the graph of environmental variables in Figure 4, it can be noted that the spike in heat loss corresponds to the moment when the fans are turned on. This lowers body temperature as shown in Figure 5, which causes vasoconstriction and lowers the rate of heat loss. Note also that the rate of heat loss was dropping just before the spike, which indicates that the fans were turned on at the right

time. These and other conclusions can be drawn by analyzing comparing the graphs generated by the program.

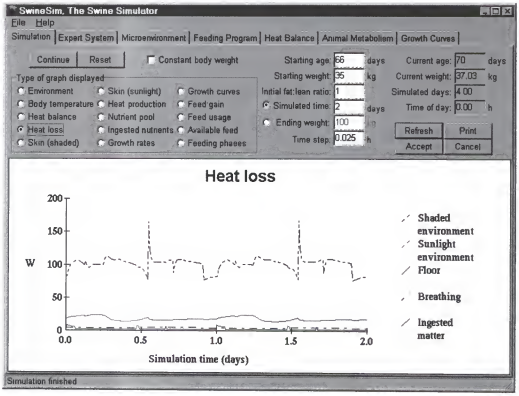


Figure 7. SwineSim simulation of heat loss partitioning.

The graph in Figure 8 shows the partitioning of the rate of heat loss through the skin to a shaded environment (W / m^2) into convection, radiation and evaporation of water at the skin surface. The rate of heat loss by convection is strongly related to air speed, shown in Figure 4. When heat loss by convection decreases, skin temperature increases and consequently radiation loss increases. The rate of evaporative heat loss is related to body temperature. As body temperature increases, the animal wallows more and therefore increases the evaporative component.

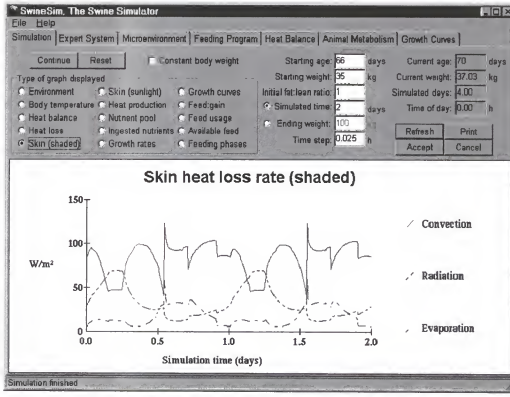


Figure 8. SwineSim simulation of heat loss through skin exposed to a shaded environment.

In Figure 9 is a graph similar to that in Figure 8, except that the skin exposed to sunlight is being analyzed, so there is an extra component for heat gain by solar radiation. As shown in the graph, this component represents a great amount of negative heat loss, which increases the temperature of the skin exposed to sunlight, as can be seen in Figure 5. This in turn increases the evaporative component of heat loss. However, since the area of skin exposed to sunlight is zero, this graph is not very useful for this example, although it does show the partitioning of heat loss, should some portion of skin become exposed to the sun.

The graph in Figure 10 shows the partitioning of heat production (W) into maintenance heat, additional heat to maintain body temperature under cold conditions,

heat generated by digestion of feed, and heat generated by lean and fat tissue deposition. In this example, the maintenance heat generation was about 80 W, and some 50 W more were necessary for tissue deposition. Heat is generated by digestion after every meal. In this example, feeding was restricted, so the simulated pigs ate three times a day. At the end of the fasting period, tissue deposition stopped due to the lack of available nutrients, so total heat production was reduced. In order to keep body temperature constant, the pig had to generate some amount of extra heat during this period, which came from tissue catabolism.

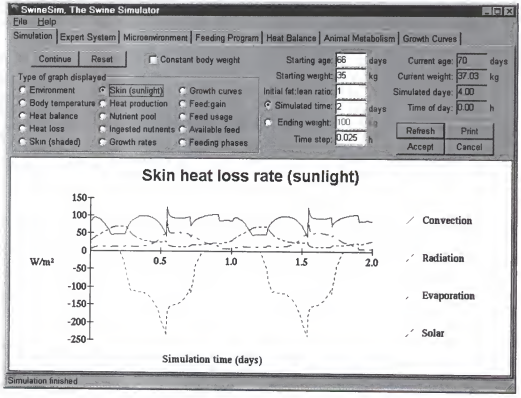


Figure 9. SwineSim simulation of heat loss through skin exposed to sunlight.

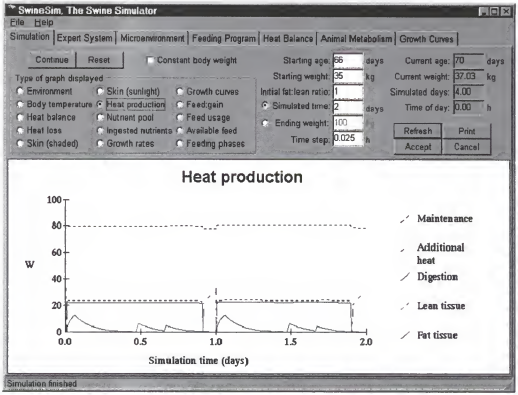


Figure 10. SwineSim simulation of heat production partitioning.

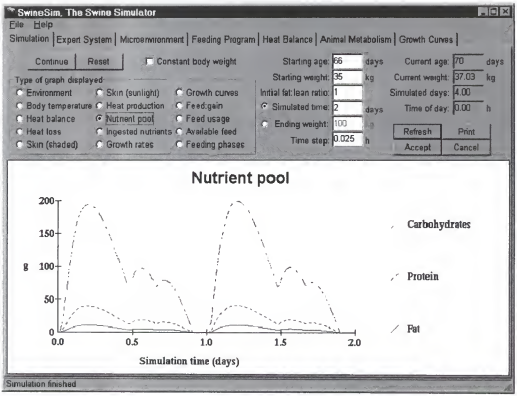


Figure 11. SwineSim simulation of nutrient pool contents.

The amount of nutrients (g) in the nutrient pool is shown in Figure 11. All the nutrients in the nutrient pool vector (carbohydrates, protein and fat) are represented in this graph. Notice that the amount of nutrients drops to zero at the end of the day, due to the feeding restriction. This corresponds to the period in Figure 10 where heat production for growth dropped to zero and additional heat had to be generated to maintain body temperature.

The amounts of carbohydrates, protein, fat and fiber in the digestive tract (g) are shown on the graph in Figure 12. The rapid increases in the quantities of these nutrients correspond to the feeding events, while the gradual drops correspond to digestion. If this graph is compared to the one in Figure 11, it can be seen that feeding events are triggered

by a reduction in the amount of nutrients in the nutrient pool, and that the nutrient pool levels increase as digestion of the nutrients in the digestive tract occurs.

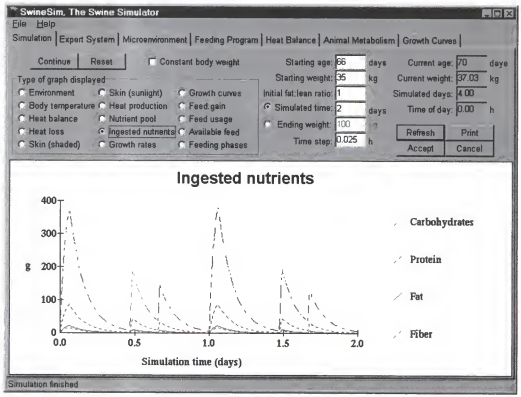


Figure 12. SwineSim simulation of digestive tract contents.

Figure 13 shows the graph with the lean and fat tissue growth rates (kg / day). In this example, the simulated pig is depositing tissue at a rate of around 0.12 kg / day of fat and slightly less lean tissue. However, at the end of the simulated day, there is no more feed available due to the restriction program, so the animal must catabolize tissue to supply energy and nutrients for maintenance. As seen in the graph, the amount of catabolized fat is much greater than the catabolized lean, so the net rate of lean deposition ends up being greater than that of fat.

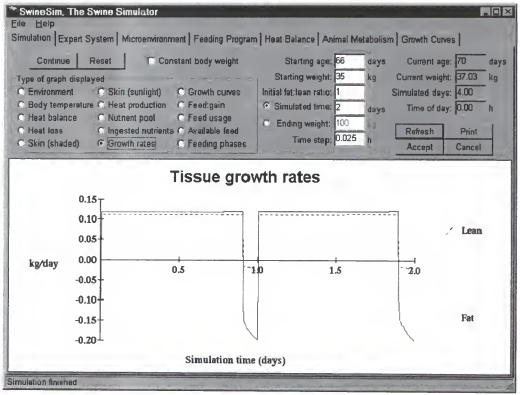


Figure 13. SwineSim simulation of tissue growth rates.

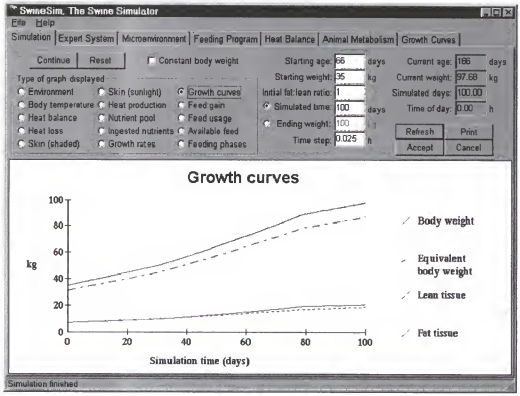


Figure 14. SwineSim simulation of growth curves.

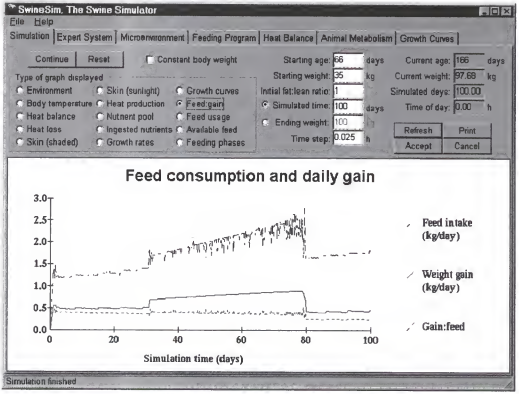


Figure 15. SwineSim simulation of feed consumption and daily gain.

Figure 14 shows the simulated growth curves for the same 100 day period shown in Figure 6. Body weight (kg), equivalent body weight (kg), and lean and fat tissue mass (kg) are represented in this graph. The middle portion of the growth curves, where the slope is higher, corresponds to a period of unrestricted feeding, while in the other segments the amount of daily feed was limited.

Feed consumption (kg / day) and growth (kg / day) in the last 24 h are plotted against time on the graph in Figure 15, for a 100 day period. The feeding restriction program can be clearly observed here, along with the corresponding reduction in weight gain. The gain : feed ratio is also plotted and, as with real pigs, it decreases with time.

Gain : feed was reduced during the periods of feed restriction because a proportionally greater portion of the ingested energy and nutrients was being used for maintenance.

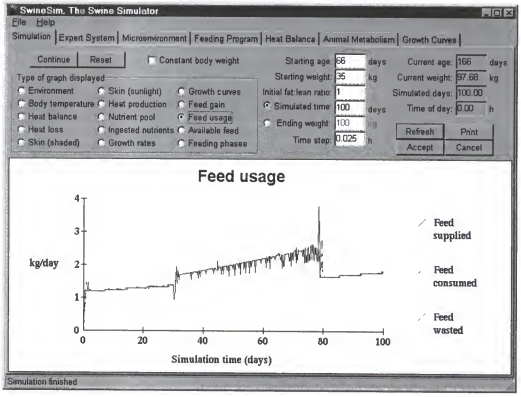


Figure 16. SwineSim simulation of feed usage.

Figure 16 shows the amount of feed supplied, feed consumed and feed wasted (kg / day). The feed consumed is the same as in Figure 15. Since the feed use efficiency factor was set to 1, no feed was wasted and, except for short time fluctuations, the amount of feed supplied was equal to the feed consumed.

Figure 17 Shows the amount of feed in the feeder, the available feed and the current feeding level (kg). Again, since the feed use efficiency factor was set to 1, feed in the feeder is equal to available feed. The amount of available feed decreases during the day

every time the animal eats, starting from the current feeding level and going down to zero. If this graph is compared to the one in Figure 11, it can be seen that when the energy level in the nutrient pool drops at the end of the day there is no feed available, so the animal does eat and stops growing.

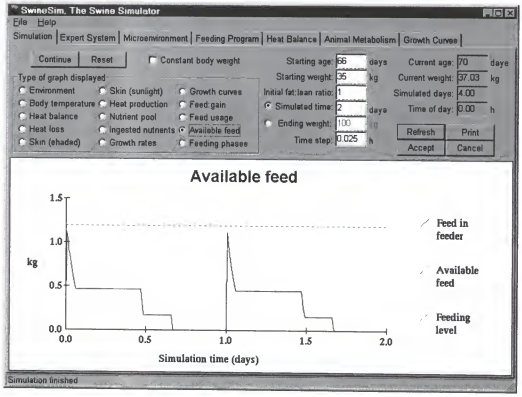


Figure 17. SwineSim simulation of available feed.

Figure 18 shows the animal's simulated body weight (kg) and the corresponding feeding phase for the 100-day trial. As seen in this graph, three feeding phases were defined in the feeding program. The first one was from 0 to 30 days, the second from 30 to 78 days and the third from 78 days on.

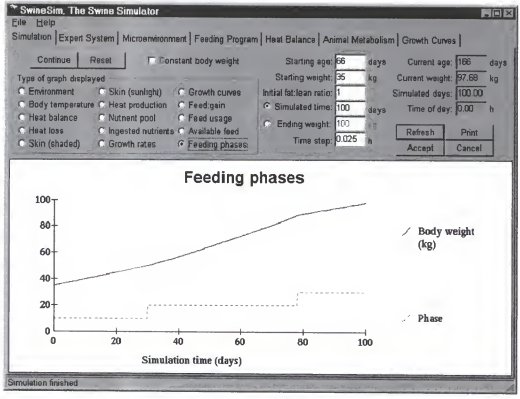


Figure 18. SwineSim simulation of heat loss partitioning.

After the simulation finishes, or every time it stops (but not when it pauses) the expert system contained in SwineSim is executed. The output of the expert system can be viewed by selecting the 'Expert System' tab in the program's main window. A sample output is shown on Figure 19. As the expert system is executed the value of each parameter set is displayed. If setting a parameter causes a rule to be executed, the user is informed of the execution, and any output resulting from the rule's execution is displayed. In the example given, one of the two currently defined rules of the expert system knowledge base was fired. It indicated that during some period in the simulation the environment was cold and the animal had to generate additional heat to maintain body temperature stable.

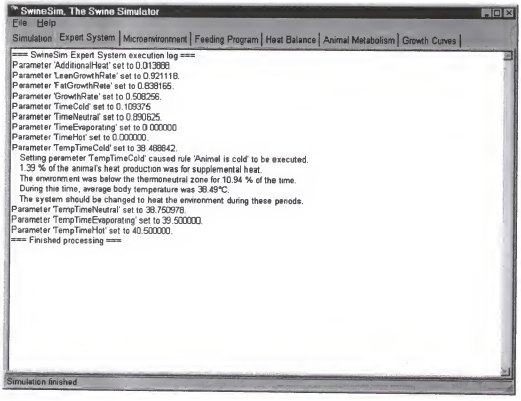


Figure 19. SwineSim Expert System module output.

The parameters of the Microenvironment module can be changed by the user in the 'Microenvironment' tab. The data input screen is shown in Figure 20. The user can immediately visualize the effect of changing the parameters for the cyclic environmental variables (air, radiant and floor temperatures, humidity, air speed and solar radiation) by clicking the 'Refresh' button. The graph in the lower portion of the screen is similar to the graph of simulated environmental variables described earlier, shown on Figure 4, and shows the variation in the cyclic variables during the course of the day. A fixed value for feed temperature can be entered by the user or the option to have feed at the same temperature as air can be selected.

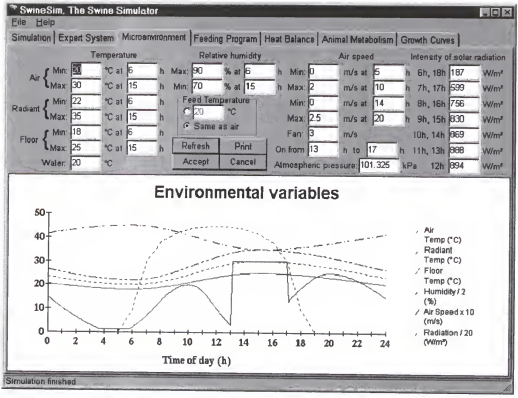


Figure 20. SwineSim Microenvironment module data input.

The data input for the Feeding Program module is done via the 'Feeding Program' tab, shown in Figure 21. Any number of diets can be entered in the grid on the middle portion of the screen. For each diet, the user specifies the amount of protein (g / kg), fat (g / kg), carbohydrates (g / kg), fiber (g / kg) and the specific volume of ingested feed in the digestive tract (cm^3 / kg). To change the name of the current diet, enter the new name in the name field, on the upper left portion of the screen and click on the 'Rename' button. To add a diet at the end of a list or to insert a diet before the diet being edited, type the name of the new diet and press the 'Add' or 'Insert' button. To delete the diet being edited from the list, press the 'Delete' button. The number of diets is automatically

updated to reflect the number of diets on the list. If the user attempts to add or insert a diet that already exists, the program selects that diet and does not add a new one.

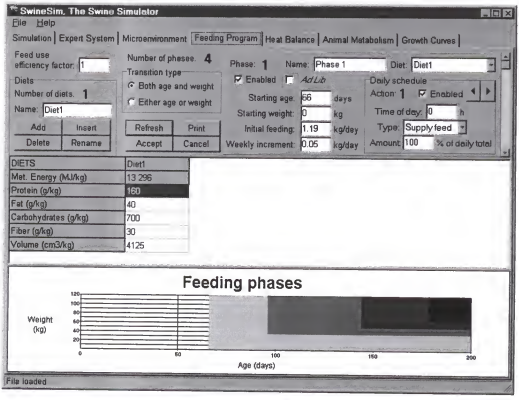


Figure 21. SwineSim Feeding Program module data input with phase transition depending on both age and weight.

After the diets have been defined, the user can select the diet to be used for each feeding phase from the diet list. Feeding phases are selected by moving the scroll bar at the upper right edge of the screen. For each feeding phase, the user can select the name of the phase, enable or disable the phase, specify *ad lib.* feeding or not, and specify the starting age and weight. If *ad lib.* feeding is not selected, the user must also supply the initial feeding level and weekly increment, as defined in the Feeding Metabolism module.

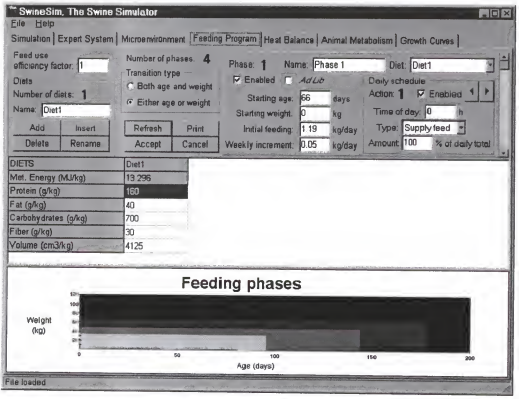


Figure 22. SwineSim Feeding Program module data input with phase transition depending on either age or weight.

Up to 10 feeding phases can be defined, each with its own feeding schedule. Each action in the feeding schedule can be enabled or disabled. If the action is enabled, the time of day and the type of action must be specified. If the action is of type 'Level feed' or 'Supply feed' and feeding is not *ad lib.*, the amount must also be specified, as a percentage of the current feeding level. Actions within a feeding phase are selected using the set of arrows on the upper right corner.

The graph on the lower portion of the screen maps the transitions from one feeding phase to another, as a function of body weight and age. If the animal's growth curve is plotted on top of this graph and the growth curve is followed, it will intercept the exact transition points from one phase to another. The transitions can be specified as occurring

when both minimum age and weight are reached by the animal, as shown in Figure 21. Alternatively, the transition can be made when either minimum age or minimum weight are satisfied. The graph in this case would be of the type shown in Figure 22

The transition type can be chosen by selecting one of the radio buttons on the upper part of the screen. The feed use efficiency factor determines the fraction of the supplied feed that is actually consumed by the animal, the rest being wasted, and it should be a number from 0 to 1.

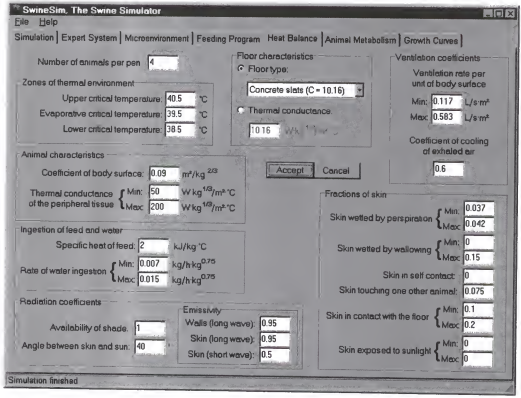


Figure 23. SwineSim Heat Balance module data input.

The parameters for the Heat Balance module can be changed by the user in the 'Heat Balance' tab, shown in Figure 23. All these parameters are described in the Heat

Balance module. The thermal conductance of the floor can either be entered directly or a type of floor can be selected from the pre-defined list.

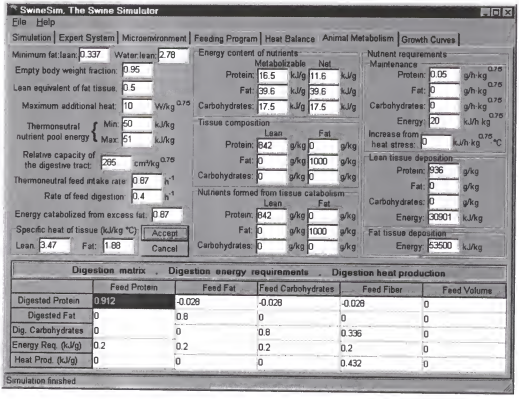


Figure 24. SwineSim Animal Metabolism module data input.

Input data for the Animal Metabolism module can be edited in the 'Animal Metabolism' tab, shown in Figure 24. All the parameters are defined in the Animal Metabolism module of the simulation model. The grid on the lower portion of the screen contains the nutrient digestion matrix ($\mathbf{D_N}$) on the top portion, the digestion energy requirement vector ($\mathbf{D'_E}$) on the next to last line and the digestion heat vector ($\mathbf{D'_Q}$) on the last line.

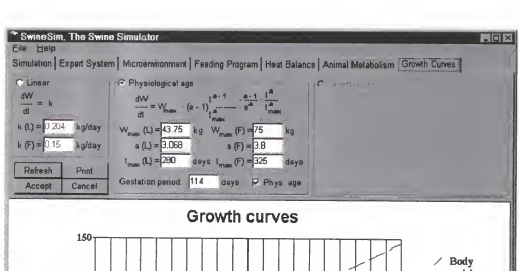


Figure 25. SwineSim growth curves data input.

The growth curves are defined in a separate tab, shown in Figure 25. As described in the implementation of the model, two growth curves are currently defined, and the user may select which one will be used. Once selected, the parameters for the curve con be defined. Two sets of parameters must be entered for each curve: one for lean tissue growth and the other for fat tissue growth. There is still space for a third growth function, for future expansions of the model. In the physiological age growth curve, chronological age may be selected instead by not checking the physiological age checkbox.

The graph on the lower portion of the screen plots the body weight, lean tissue and fat tissue growth curves for the currently defined parameters. After the user changes the parameters, this curve can be updated by clicking on the 'Refresh' button.

The program is divided into units, each one responsible for part of the tasks of the program. The Main unit defines a class for the main window and a corresponding object. It also defines all the components of the main window and the event handlers associated with it. The Data Transfer unit defines a class for dealing with data transfer between the graphical user interface (GUI) and memory and between memory and a file on disk. It is also responsible for initializing all the variables to their proper values and storing the simulation results into a file for any possible further processing. The Matrix unit defines classes for vectors and matrices, and their associated mathematical operators.

The Simulation unit defines the simulation class, which controls the initialization and the execution of time steps, as well as the storage of the simulated results in memory. This unit also contains functions from the data transfer class which control the flow of simulation parameters between the GUI, memory and disk. It also takes care of displaying the simulation graph and controlling the main window's idle routine. When the simulation starts, the parameters are initialized and a flag is set to indicate the simulation is running. When the computer is idle, it calls the main window's idle function, which in turns tests the value of the simulation running flag, executing one time step if the flag is set. This way, the computer is able to run the simulation model on the background, while performing other tasks.

The Expert unit contains the inference engine for the expert system and is responsible for defining a class for the expert system parameters and a base class for the expert system rules. The Knowledge Base unit contains the knowledge base to be executed by the expert system, including listing of parameters and definition of rules.

The Environment, Feeding Program, Heat Balance and Metabolism units define classes and instances of each of the modules in the simulation model. Each one of these four units also contains the functions for initializing and executing a time step for the corresponding simulation modules, as well as the data transfer routines used. These units also contain functions for updating the on-screen graphs and for validating the user's input. Finally, the About Box unit takes care of displaying the program's About box.

CHAPTER 11
SYSTEM VALIDATION

Verification and Calibration

The first step for validating a model is to verify that it is indeed implemented as it was designed. Part of the validation procedure consists of debugging the program and verifying that the output is reasonable. This sometimes involves calculating values by hand and comparing the program output with the calculated results.

Not all the parameters found in the literature are believed to be correct. Some of the values are well known and their effect on the model is clear, based on the underlying logic. For example, the energy requirement for maintenance is believed to be correct. Although some variation may occur in the requirement, most of the literature seems to agree on the range of values used and whatever the true value may be it will probably not be off by much. Parameters such as the specific heat of water and the Stefan-Boltzmann constant are so accepted that they were hard-coded into the program and the user cannot change them.

Other parameter values used in the model are not that reliable. Either the values in the literature are dubious or the simplifications made in the model make the parameters have a different meaning. The portion of the model where this seems to be more evident is in the feed intake and digestion. The model oversimplifies the process, such that some

of the parameters had to be adjusted empirically. As a result, the values that made the model behave correctly do not necessarily reflect real life. Luckily, only a few parameters fall in this category, and the discrepancies may be corrected when the model is refined.

The parameters which had to be adjusted this way were the minimum (E'_{min}) and maximum (E'_{max}) thermoneutral nutrient pool energy levels. The relative capacity of the digestive tract (k_v) and the rates of thermoneutral feed ingestion (r'_i) and digestion (r_d) used were calculated assuming that the specific volume of feed in the digestive tract (v_f) was 4125 cm^3 / kg, in a way that these values depend on each other, and they should not be compared with values that may be found in the literature. Further refinement of the feed intake and digestion process may result in these parameters being more meaningful.

As a research tool, an option is included in the simulation model to keep body weight constant. Using this option will reset lean and fat tissue weight to their original values at the end of each time step. Growth rates are still calculated correctly, for that fixed body weight. This allows the simulation to proceed indefinitely, and is useful for reaching steady state conditions after a change in one of the simulation parameters. This feature was used in much of the model testing.

A 70 kg pig with a 1 : 1 initial fat : lean ratio was simulated, generally using the constant body weight option to keep the weight at 70 kg. Unless stated otherwise, feeding was *ad lib.* with no restrictions. The feed used had 18 % protein, 6 % fat, 5 % fiber and 68 % carbohydrates, and each kg of feed was assumed to occupy 4125 cm^3 in the digestive tract. The specific heat of feed was assumed to be 2 kJ / kg · °C. The environment was initially set to a constant 20 °C for all temperatures, 80 % relative humidity, no solar radiation and constant air speed of 0.5 m / s. Under these conditions,

the animal was within the thermoneutral zone. Except where noted, the default values listed in the model description were used for all the parameters.

Time Step and Nutrient Pool Energy Threshold Interval

The first variable to be calibrated is the time step used in the simulation process. An initial estimate of 0.1 h was used, but it was not always adequate. In order to determine what time step to use, it was necessary to monitor a variable that was affected by it. The number of daily meals was found to depend greatly on the difference between the maximum (E'_{max}) and minimum (E'_{min}) thresholds for the energy concentration in the nutrient pool (E_c). If this difference was small, the number of daily meals was also affected substantially by the time step.

Both the time step and E'_{max} were varied while keeping E'_{min} fixed at 50 kJ / kg, and the resulting number of daily meals was measured. The results obtained are presented in Table 1.

The true value for the number of feeding events would be that obtained with an infinitesimally small time step, if no round-off error occurred. For practical purposes, the time step of 0.001 h was assumed to be correct. For E'_{max} equal to E'_{min}, the number of meals is theoretically infinite, but because time steps introduce an error, it becomes finite. As shown on Table 1, larger time steps introduce more error and make the number of meals smaller. This happens because the model assumes that the animal eats for at least Δt amount of time, so increasing Δt might make the simulated pig eat more than it needs, therefore having more reserves in the digestive tract and not needing to eat again so soon.

Table 1. Number of daily meals as a function of the time step and the nutrient pool energy threshold interval.

Nutrient pool energy threshold (E'_{max}, kJ/kg)*	Time step (Δt, h)								
	0.001	0.002	0.005	0.01	0.02	0.025	0.05	0.1	0.2
50.000	59.0	37.0	24.0	17.0	13.0	11.0	8.0	6.0	4.5
50.005	32.0	29.0	22.0	16.0	12.0	11.0	8.0	6.0	4.5
50.010	26.0	25.0	20.0	16.0	12.0	11.0	8.0	6.0	4.5
50.020	22.0	21.0	18.0	15.0	12.0	11.0	8.0	6.0	4.5
50.050	16.0	16.0	15.0	13.0	11.0	10.0	8.0	6.0	4.5
50.100	13.0	13.0	12.0	11.0	10.0	9.5	7.0	6.0	4.5
50.200	11.0	11.0	10.0	9.5	8.5	8.0	7.0	6.0	4.5
50.500	8.0	8.0	7.5	7.0	7.0	6.5	6.0	5.0	4.5
51.000	6.1	6.1	6.1	6.0	5.8	5.6	5.2	4.6	3.6
52.000	4.9	4.9	4.8	4.8	4.7	4.7	4.4	4.0	3.2
55.000	3.6	3.6	3.6	3.6	3.5	3.5	3.4	3.2	3.0
60.000	2.8	2.8	2.8	2.8	2.8	2.8	2.7	2.6	2.5

* Maximum; the minimum (E'_{min}) was fixed at 50 kJ/kg.

Increasing the difference between E'_{min} and E'_{max} decreases the number of meals, because the animal truly needs some time to make the nutrient pool reserves increase to the new level. During this time, the animal will be eating, increasing the reserves in the digestive tract. After the animal stops eating, it will continue digesting, therefore increasing the nutrient pool reserves beyond E'_{max}. If the difference between E'_{min} and E'_{max} is larger, the animal will have eaten more and the E_e peak will be higher.

As the number of daily meals is decreased, the sensitivity of the model to the time step also decreases, because the animal will be eating for a larger number of time steps. For E'_{max} equal to 60 kJ / kg, the effect of time step is very small. The computing time to run the model is inversely proportional to the time step. Although small time steps are

more accurate, it is impractical to use very small ones. A compromise must be reached between computing time and accuracy.

McDonald *et al.* (1991) measured an average of 5.5 meals per day, which is roughly what is obtained by setting E'_{max} to 51 kJ / kg. At this level, a time step of 0.025 h (1.5 minutes) seems to be adequate, with the reduction in the number of daily meals being only from 6.1 to 5.6. Using this time step, the model will still respond adequately to changes in E'_{max} and E'_{min} and, although it runs four times slower than originally planned, still can be executed within a reasonable amount of time. Gross approximations of the results can be obtained by using larger time steps, but the number of meals and possibly other parameters will probably be incorrect.

Based on these results, the time step and the energy pool threshold interval were set to 0.025 h and 1 kJ / kg, respectively.

Nutrient Pool Energy Threshold Level

The criteria used to calibrate parameters related to feed intake were the frequency of feeding events, the nutrient levels in the nutrient pool and the response of the simulated pig to a fasting period. Since the only effect that heat stress has on production is to reduce or halt feed intake, it is important to evaluate how the simulated pig will react to a certain period of feed withdrawal. Ideally, fasting for a couple of hours should not interfere with production. However, in order to adequately simulate the effects of heat stress on growth, the simulated pig should respond to longer periods of fasting by reducing growth and even losing weight.

Table 2. Feed intake, number of meals and fasting time to cause reduction in growth for various nutrient pool energy threshold levels.

Nutrient pool energy thresholds (kJ/kg)		Daily	Feed intake	Fasting reserves	Daily meals
E'_{min}	E'_{max}	meals	(kg/day)	(h)	when fasting
0	1	7.0	2.0	0.0	7.0
1	2	5.6	2.0	2.1	5.0
2	3	5.5	2.0	1.9	5.0
5	6	5.5	2.0	4.0	5.0 *
10	11	5.5	2.0	2.7	4.3 †
15	16	5.5	2.0	1.9	4.3 †
20	21	5.5	2.0	5.1	4.0
21	22	5.5	2.0	5.0	4.0
22	23	5.5	2.0	4.8	4.0
22.5	23.5	5.5	2.0	4.8	3.5 ‡
23	24	5.5	2.0	4.8	3.0
25	26	5.5	2.0	4.5	3.0
30	31	5.5	2.0	4.0	3.0
40	41	5.5	2.0	3.7	3.0
50	51	5.5	2.0	6.7	3.0 *
60	61	5.5	2.0	6.2	3.0 *
70	71	5.5	2.0	5.8	2.0
80	81	5.5	2.0	5.5	2.0
90	91	5.5	2.0	9.7	2.0 *
100	101	5.5	2.0	9.5	2.0 *
120	121	5.5	2.0	9.2	2.0 *
150	151	5.5	2.0	8.7	1.0
156	157	5.5	2.0	8.6	1.0
157	158	5.5	2.0	21	1.0 *
200	201	5.5	2.0	21	1.0 *
250	251	5.5	2.0	21	1.0 *
300	301	5.5	2.0	21	1.0 *
400	401	5.5	2.0	21	1.0 *
500	501	5.5	2.0	21	1.0 *
1000	1001	5.5	2.0	21	1.0 *

* The last meal was interrupted by the removal of the feed

† One out of every three days had a fifth meal, which was interrupted by removal of feed

‡ One out of every two days had a fourth meal

E'_{min} was set to different values, with E'_{max} always being 1 kJ / kg above, and the resulting number of meals per day and meal size was recorded as this value changed. After these values were recorded, a fasting period was simulated, in which feed was removed for a certain period of time. The duration of this fasting period was changed until a visible effect on growth was observed every day. For each simulation, the model ran until it reached a dynamic equilibrium condition in the nutrient pool before the resulting output was evaluated. The constant body weight option was used so that the animal would always be 70 kg. The minimum fasting period that caused growth reduction was recorded, with a 0.1 h accuracy. The results are presented in Table 2.

The energy threshold had no effect on the number of meals and on feed intake, except when it was unreasonably small. By dividing daily feed intake by the number of meals, the average meal size can be calculated as 364 g. McDonald *et al.* (1991) measured an average of 5.5 meals per day, with an average duration of 12.5 minutes. Considering an average eating rate of about 1 g / min $\cdot$ $kg^{0.75}$, feed intake per meal for a 70 kg pig would be 303 g. These values are very close to the simulated results, and the model was considered to be simulating the number of meals and meal size satisfactorily. However, pigs tend to eat more during the day (McDonald *et al.*, 1991), and the model does not account for that behavior.

If the data is grouped according to the number of uninterrupted meals, there is a decrease in the fasting time as E'_{max} increases. This can be explained if the feeding behavior is analyzed. After a period of fast in which growth is affected, the nutrient pool reserves are at zero. This causes the animal to immediately begin eating as soon as feed is supplied. The animals in which the threshold is set higher will eat longer. As a result, the

digestive tract will have more feed and the animal will take longer to consume the second meal. This will cause a time shift in the meal schedule, which will be more pronounced if E'_{max} is higher. For the same number of meals, a longer delay means the feed must be available for a longer time, therefore the fasting period must be shorter.

A smaller number of daily meals means the meals will be longer and more feed will be ingested per meal. For this reason, the maximum fasting period increased with larger threshold values whenever a reduction in the number of meals was observed. In the last few threshold values (which had a fasting time of 21 h) one single meal was sufficient to provide nutrients for the entire day. However, the animals needed over 3 hours to consume that single meal. This occurred because the digestive tract became full and the feed intake capacity was limited by the rate of digestion. Although the simulated pig was constantly eating during these three hours, the feeding rate was reduced. The nutrient pool energy level never reached the threshold, and the maximum energy in the pool was determined by the animal's capacity to ingest feed. Under the limiting fasting conditions, the threshold could have been increased indefinitely without any effect on the rest of the system. All the threshold levels below that limit required at least two meals to avoid reduction in growth (the number of daily meals when fasting shown on the table refers to the situation in which growth is affected).

Based on these results, a value was chosen for the threshold levels. Values above 150 kJ / kg could result in only one meal per day, which was not desired. If less than 20 kJ / kg was used, the nutrient reserves would be too small and the animal would be too sensible to fasting. The intermediate values of 50 and 51 kJ / kg were chosen for E'_{min} and E'_{max}, respectively.

Another simulation experiment was run in which the threshold level was set to 5000 kJ / kg and the simulation ran until the nutrient pool reached steady state. At this point, feed was removed and the amount of time to consume the nutrient reserves was recorded as 11.5 days, which means the simulated pig consumed the nutrient pool reserves at a rate of 18.1 kJ / kg · h. Theoretical limits for the maximum feed withdrawal time with no reduction in growth would be between the time to exhaust the reserves in the nutrient pool (assuming the animal was about to eat) and this value plus the average time between meals (assuming the animal just ate). For the 50 kJ / kg threshold, the maximum feed withdrawal time would be between 2.8 and 7.1 hours, which seems to be a reasonable range. The same experiment was conducted for a 20 kg pig, using the same parameters, except environmental temperature, which was raised to 25 °C to keep the animal within the thermoneutral zone. At that size, the simulated pig ate an average of 6.1 meals per day and daily feed consumption was 0.8 kg. The nutrient reserves were consumed in 8.15 days, which is equivalent to an energy usage of 25.6 kJ / kg · h. With these values, the maximum feed withdrawal time decreased to a range of 2.0 to 5.9 hours.

Validation

In order to validate the model, an experiment was replicated using the simulation model. The field experiment was described by Nienaber *et al.* (1990b), and had the objective of verifying the carcass composition and maintenance energy requirements of swine growing at different weight gain rates. This data set was used because it contained not only growth, but also carcass composition and heat production data.

Sixty-six pigs with a starting weight around 35 kg and gaining 0.5 kg / day were used in the experiment. The experiment consisted of three treatments involving restrictions in feeding. In the control treatment (C), the pigs grew 0.5 kg / day for 112 days, until they reached 91 kg. In the medium growth treatment (M), the pigs were kept at 35 kg live weight for 30 days and then grew 0.68 kg / day for 82 days, until they reached approximately 91 kg. In the high growth treatment (H), the pigs were kept at 35 kg live weight for 30 days, then grew 1 kg / day for 30 days until they reached 65 kg, and finally grew 0.5 kg / day for the remaining 52 days, until they reached 91 kg.

These pigs were assigned to 11 slaughter groups within the treatments. One group of animals was slaughtered at the beginning of the experiment, and represented initial body composition of the animals in all three treatments. The second group of animals was slaughtered after 30 days at 35 kg, and represented the body composition of treatments M and C after the period of zero growth. The remaining nine slaughter groups were animals from each of the three treatments, at the time they reached 50, 65 and 91 kg.

Daily feed was given in one meal each day, with minimum wastage. Room temperature was 23 °C and the feed had 16 % protein and 13.3 MJ / kg metabolizable energy. The feed in the simulation was set to 16 % protein, 4 % fat, 70 % carbohydrates and 3 % fiber, which resulted in 13.3 MJ / kg metabolizable energy, and the specific volume in the digestive tract was set to 4125 cm^3 / kg.

The 35 kg pigs slaughtered at the beginning of the experiment had 5.63 kg of fat and 4.74 kg of protein, which, for the current values of tissue composition, corresponds to 5.63 kg of lean tissue, so the initial fat : lean ratio was set to 1. In order to predict the maximum rate of growth of animals of this size, *ad lib.* feeding was used in a simulation

run with the constant weight option set. The model predicted an instantaneous growth rate of 0.553 kg / day. Since the animal was within the thermoneutral zone and no feeding restrictions were applied, this value is completely dependent on the growth curve used.

Based on this result alone, the experiment cannot be replicated using the currently defined growth function. Since the growth rate is a function of physiological age, which is a function of lean mass, animals with 35 kg of body weight will not be able to gain 1 kg / day, as defined in the experimental design. The growth functions used have no provision for compensatory growth, therefore the simulated pig will grow less than the real pig. This is not a fault in the simulation model, but in the definition of the growth functions, which, as described, are independent of the model. The rate of gain will increase if the pig is kept at 35 kg for 30 days, if carcass composition changes such that the ratio of fat : lean tissue is decreased, although this change alone would not be sufficient to practically double the rate of weight gain.

To verify this assertion, the constant weight option of the model was turned off and 30-day simulation runs were executed, varying the amount of feed supplied until body weight remained approximately constant. By restricting feed intake to 0.61 kg / day, the simulation model predicted a final weight after 30 days of 35.04 kg. During this period, lean tissue mass in the simulated pig increased from 6.96 to 7.15 kg, and fat tissue mass decreased from 6.96 to 6.27 kg, which reduced the fat : lean ratio from 1 to 0.877. A similar trend was observed in the real pigs. The animals slaughtered after 30 days of zero growth had 4.88 kg of fat and 6.02 kg of protein, which corresponds to 7.15 kg of lean tissue and a fat : lean ratio of 0.683. The real pigs had a higher fat reduction than the

simulated pigs, which indicates that the fraction of energy catabolized from excess fat (k_E) may be more than 0.5, as assumed in the model.

If k_E is changed to 1, assuming all energy is catabolized from fat, feed intake must be reduced to 0.4 kg / day, in order to keep the animal at constant weight. This occurs partly because fat tissue has more energy than lean tissue, so a smaller amount of fat needs to be catabolized to supply energy during the periods when the animal is fasting. The other reason is that body water is associated with lean tissue, so the actual body weight reduction by a decrease in lean tissue is much greater than by a reduction of an equal amount of fat tissue. During restricted feeding, the animal alternates periods of weight gain (after feeding) with weight loss (at the end of the fasting period). When k_E is increased, the animal loses less weight to make available the same amount of energy. If feed intake remains the same, the weight gain after feeding will remain the same, and therefore be greater than the weight loss. To compensate for that the amount of feed supplied must be reduced.

The change of k_E to 1 and the restriction in feed intake to 0.40 kg / day resulted in a final weight of 34.99 kg after 30 days. During this period, lean tissue mass in the simulated pig increased to 7.61 kg, and fat tissue mass decreased to 4.47 kg, which reduced the fat : lean ratio to 0.588. This reduction was more than what was observed in the real pigs, which shows that, as expected, at least some amount of lean tissue is catabolized to generate energy during the periods of deficiency. By successive approximations, values of 0.87 for k_E and 0.47 kg / day for feed intake were determined. Using these values the simulated pig weighed 34.94 kg after 30 days, and had 7.44 kg of

lean tissue and 5.07 kg of fat tissue, which resulted in a fat : lean ratio of 0.681, which only differs 0.3 % from the observed results.

By setting the constant body weight option and running the simulation for one more day with *ad lib.* feeding, after the simulated pig was kept at 35 kg for 30 days, the maximum rate of weight gain obtained was 0.576 kg / day. This represents an increase of only 4.2 % due to the change in body composition, and is nowhere near the 1 kg / day proposed in the experiment. Again, this is due to the fact that the growth curve used, based on physiological age, does not account for compensatory growth.

One way to partially correct this discrepancy is to use chronological age instead of physiological age to determine the growth potential, so that the growth rates are a function of age and not lean mass. By comparing the growth rates at different ages with the initial growth rate (0.553 kg / day), the initial physiological age of the pigs was determined to be 66 days, at which time the potential weight gain was 0.554 kg / day. The initial age of the animals was set to this value, the constant body weight option was turned off, and a 30-day run was simulated, with 0.48 kg / day of feed intake. After this period, the simulated pig weighed 35.02 kg, had 7.45 kg of lean tissue, 5.10 kg of fat tissue, and a fat : lean ratio of 0.684.

The maximum rate of weight gain at 96 days of age was 0.711 kg / day, which is higher than the 0.68 kg / day of the medium growth treatment, although not sufficient to simulate the high growth treatment. Because it accounts for some degree of compensatory growth, chronological age was used in the growth functions. As an alternative to the high growth treatment, it was decided to have the animals for that treatment on unrestricted feeding until they reached the weight of the control group.

The amount of feed necessary to obtain the growth rates described for the control, medium growth and high growth treatments were determined for each period. This was done by setting the feed supply in the feeding program to different levels and observing the resulting output. When the final weight at the end of each week was approximately equal to the goal weight, the feeding level was considered to be correct. The resulting values for initial feeding and weekly increment to be added to the initial value are shown in Table 3.

Table 3. Feeding rates required to obtain different simulated growth rates.

Treatment*	Period (days)	Growth rate (kg/day)	Initial feed (kg/day)	Weekly increment (kg/day)
C	0-30	0.500	1.190	0.050
C	30-60	0.501	1.400	0.040
C	60-112	0.502	1.580	0.045
M, H	0-30	-0.001	0.460	0.010
M	30-60	0.685	1.560	0.055
M	60-112	0.683	1.795	0.060
H	30-78	0.814	Ad lib	
H	78-112	0.498	1.650	0.055

* C = Control; M = Medium growth; H = High growth

The objective was to reach 91 kg after 112 days, starting with a 35 kg pig, using each of the schemes described earlier. Feeding in treatment H was *ad lib.* until these animals reached the weight of the controls. This happened at day 78, when the pigs weighed 74.05 kg, and feeding from that point on was restricted to obtain a 0.5 kg / day growth rate. During the period of *ad lib.* feeding, the simulated pigs of treatment H gained

0.814 kg / day. As shown in Table 3, a weekly increment in feed supply was necessary, to account for the increase in the maintenance requirements as the animals grew larger. Even for the zero growth period a weekly increase in feeding was necessary, to compensate for the greater maintenance requirements of a leaner pig. Body weight as a function of time for each of the three treatments is shown in Figure 26, and the feed intake used to achieve these body weights is shown in Figure 27.

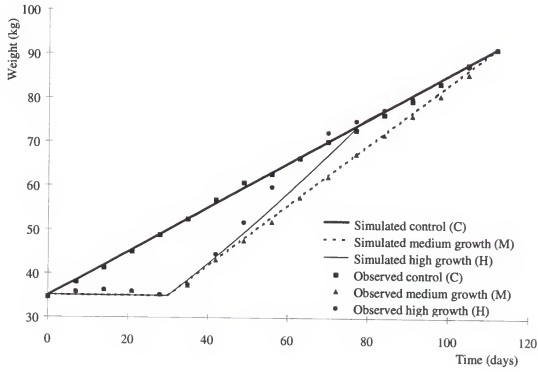


Figure 26. Simulated and observed growth curves for pigs under different growth rates.

Simulated feed intake was generally less than the observed values, in part because not all the parameters in the model are correctly calibrated. The observed and simulated curves tend to follow each other in a parallel fashion. Note that neither represent

voluntary feed intake, except for the middle portion of the simulated high growth treatment. In both cases, the feed intake was calculated to obtain a certain rate of growth. Figure 27 also demonstrates the workings of the Feeding Program module, which was responsible for controlling the amount of feed supplied to the animal.

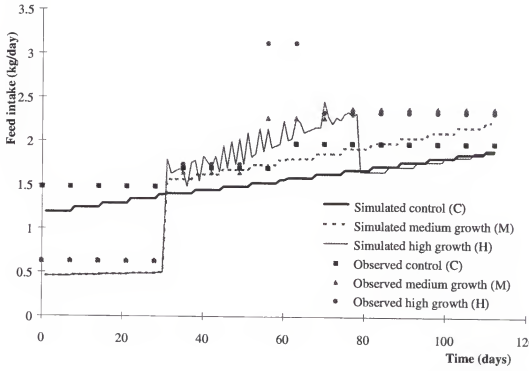


Figure 27. Simulated and observed feed intake for pigs under different growth rates.

Growth rates and feed conversions obtained are shown in Tables 4 and 5. The simulated pig followed the intended growth curve closely, except for treatment H, in which growth from 35 to 65 kg was only 0.795 kg / day.

The simulated feed conversions were generally lower than the observed values, because less feed was consumed. However, in the animals that spent 30 days at 35 kg

body weight, observed feed conversion for the 35-50 kg range was lower than what the model simulated. Again, since the model does not account for compensatory growth, it did not predict an increase in the efficiency of growth for that period. Apart from that, the model correctly predicted that growth would be more efficient for treatment H and less for treatment C, relative to treatment M, except for the 65-91 kg period, when the efficiency of H would be reduced.

Table 4. Observed and simulated growth rates (kg / day).

Weight range	Observed*			Simulated*			(Sim - Obs)*		
(kg)	C	M	H	C	M	H	C	M	H
35-50	0.485	0.716	0.899	0.500	0.687	0.756	0.015	-0.029	-0.143
50-65	0.493	0.701	1.020	0.501	0.680	0.838	0.008	-0.021	-0.182
65-91	0.500	0.648	0.523	0.502	0.685	0.587	0.002	0.037	0.064
35-65	0.489	0.712	0.957	0.500	0.683	0.795	0.011	-0.029	-0.162
35-91	0.500	0.687	0.690	0.501	0.684	0.683	0.001	-0.003	-0.007

* C = Control; M = Medium growth; H = High growth

Table 5. Observed and simulated feed conversions (kg feed / kg gain).

Weight range	Observed*			Simulated*			(Sim - Obs)*		
(kg)	C	M	H	C	M	H	C	M	H
35-50	3.05	2.29	1.92	2.66	2.47	2.36	-0.39	0.18	0.44
50-65	3.43	3.23	3.05	3.04	2.75	2.53	-0.39	-0.48	-0.52
65-91	3.94	3.66	4.43	3.50	3.05	3.48	-0.44	-0.61	-0.95
35-65	3.18	2.73	2.54	2.85	2.61	2.45	-0.33	-0.12	-0.09
35-91	3.50	3.05	3.46	3.12	2.78	2.82	-0.38	-0.27	-0.64

* C = Control; M = Medium growth; H = High growth

The estimated values of lean mass, fat mass and fat : lean ratio for the 11 slaughter groups described earlier are shown in Table 6. When feeding was restricted, the fat : lean ratio decreased because the animal had to catabolize tissue to meet the maintenance requirements during some part of the day. After 30 days with no growth, the animals in the slaughter group 35-NG had a dramatic reduction from 1 to 0.683 in the fat : lean ratio, due to a 27 % reduction in fat mass associated with a 7 % growth in lean tissue.

Table 6. Simulated tissue mass and fat : lean ratio of pigs from different slaughter groups.

Slaughter group	Treatment*	Time (days)	Age (days)	Weight (kg)	Lean mass (kg)	Fat mass (kg)	Fat : lean
35-C	C,M,H	0	66	35.00	6.956	6.956	1.000
35-NG	M,H	30	96	34.97	7.443	5.085	0.683
50-C	C	30	96	49.99	9.963	9.832	0.987
50-M	M	52	118	50.07	10.303	8.619	0.837
50-H	H	50	116	50.08	10.277	8.732	0.850
65-C	C	60	126	65.01	12.975	12.715	0.980
65-M	M	74	140	65.03	13.113	12.208	0.931
65-H	H	68	134	65.17	13.039	12.626	0.968
91-C	C	112	178	91.09	18.093	18.146	1.003
91-M	M	112	178	91.04	17.889	18.870	1.055
91-H	H	112	178	90.98	17.944	18.607	1.037

* C = Control; M = Medium growth; H = High growth

The experimental results of carcass composition of the real animals are given in Table 7. Lean mass is calculated by dividing the result of the protein analysis by 0.842 (the fraction of protein in lean tissue), to facilitate the comparison with the simulated results. This makes lean mass in both simulated and real pigs a direct function of protein

mass. The fat : lean ratio of slaughter groups 35-C and 35-NG was identical because the first group characterized the initial conditions and the second was used in the calibration of the model. Therefore, these values should not be used to validate the model. The other nine slaughter group results, however, were not used in the calibration and can be used in the validation.

Table 7. Observed tissue mass and fat : lean ratio of pigs from different slaughter groups.

Slaughter group	Treatment*	Weight (kg)	Lean mass (kg)	Fat mass (kg)	Fat : lean
35-C	C,M,H	31.83	5.629	5.630	1.000
35-NG	M,H	33.61	7.150	4.880	0.683
50-C	C	47.28	9.216	9.010	0.978
50-M	M	45.38	8.551	8.090	0.946
50-H	H	44.22	8.729	7.700	0.882
65-C	C	59.46	11.473	12.730	1.110
65-M	M	60.76	11.152	13.250	1.188
65-H	H	57.27	10.570	12.190	1.153
91-C	C	89.40	16.960	19.850	1.170
91-M	M	87.67	15.891	20.470	1.288
91-H	H	88.13	16.793	22.460	1.337

* C = Control; M = Medium growth; H = High growth

In general, the model simulated a smaller fat : lean ratio than what is observed in the real pigs as the animals get older. There could be a few reasons for this. First, the k_E value of 0.87 may be overestimated, which would mean that the animals actually use less fat to supply energy during tissue catabolism than what is assumed by the model. Therefore, the simulated decrease in lean mass at the end of the fasting periods may have

been smaller and the decrease in fat mass larger, which would decrease the simulated fat : lean ratio, relative to the true values. Also, the growth curves may not be correct for the specific genotype of pigs used, which could easily alter the expected proportions of lean and fat.

The nutrient pool vector may be too simple, and therefore the nutrient requirements for lean growth (only protein, in this case) are always satisfied, as long as the energy requirements are satisfied. Therefore, lean tissue growth may not be as restricted in the simulation as it is in reality. Finally, the growth curves simulated by the model shift abruptly from growth (when there are nutrients in the nutrient pool) to catabolism (when the nutrients are exhausted). An alternative scheme, where the growth rate would decrease with a reduction in the nutrients in the pool may generate different results.

Nevertheless, when the values are compared between the slaughter groups there seems to be a general agreement between the simulated and the observed data, when comparing one treatment with another. This comparison is clearer in Figure 28, in which the simulated values are plotted as lines and the observed values as data points. In both data sets, the fat : lean ratio varied less in the control group, due to the fact that these pigs started out with more fat and a ratio which was closer to that of the older pigs. In the two treatments where feed was restricted at the beginning of the experiment, the starting ratio was very low, due to the consumption of fat reserves. When the animals started growing at a higher rate, however, they deposited relatively more fat than the control group, because feeding was less restricted. At the end of the experiment, the animals from treatments H and M had a fatter carcass than the animals from the control group. This was

observed in both simulated and real pigs, which indicates that the model behaves in the same way that real pigs do.

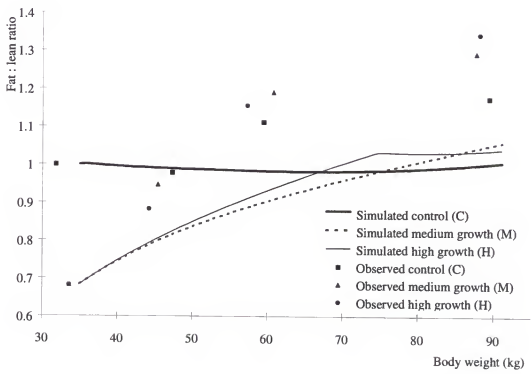


Figure 28. Simulated and observed fat : lean ratios of pigs with different growth rates.

Fasting heat production was simulated by setting the constant body weight option, removing feed and running the simulation for 24 h. The simulated values for each of the slaughter groups and the observed values from the experiment are shown in Table 8. The simulated values of fasting heat production are very similar, on a metabolic weight basis, and are also very close to the observed values. The simulated value is larger for the 35 kg pigs because some small amount of additional heat had to be produced to maintain body

temperature. When only the maintenance heat is considered, and the additional heat to maintain body temperature is excluded, the values are similar to the other weights.

Table 8. Observed and simulated fasting heat production in pigs.

Slaughter group	Treatment*	Body weight kg	Fasting heat production: Simulated W	Simulated $W/kg^{0.75}$	Observed $W/kg^{0.75}$
35-C	C,M,H	35.00	84.12	5.846	5.249
35-NG	M,H	34.97	84.09	5.848	4.199
50-C	C	49.99	98.43	5.236	5.128
50-M	M	50.07	99.76	5.300	5.191
50-H	H	50.08	99.67	5.294	5.181
65-C	C	65.01	119.94	5.239	4.582
65-M	M	65.03	120.42	5.259	5.152
65-H	H	65.17	120.27	5.243	5.752
91-C	C	91.09	154.18	5.229	4.838
91-M	M	91.04	153.51	5.208	5.152
91-H	H	90.98	153.64	5.216	5.128
35-C†	C,M,H	35.00	75.26	5.230	
35-NG†	M,H	34.97	77.24	5.372	

† Excluding heat production to maintain body temperature

* C = Control; M = Medium growth; H = High growth

Basal maintenance was reduced in the pigs from group 35-NG, but this was not observed in the simulated pigs. In fact, the simulated pigs of this group showed a slight increase in fasting heat production, due to the greater proportion of lean tissue, which was assumed to have a higher metabolic rate than fat. The reason for the lower heat production rate in group 35-NG, according to Nienaber *et al.* (1990b) is the fact that

animals that are not growing have less intense basal metabolism, a fact that is not accounted for in the model.

Bond *et al.* (1959) mention values of total heat loss in pigs which vary from about 115 W for a 35 kg pig to 172 W for a 91 kg pig. These values are greater than the ones simulated in the present experiment and the ones observed by Nienaber *et al.* (1990b). This may have occurred for three reasons. First, the values of Bond *et al.* (1959) include evaporation of moisture from urine, which is important for the design of buildings, but not from the point of view of heat loss from the pigs. Second, the pigs used by were not fasting, and therefore were generating more heat than the simulated fasting animals. Third, the genotype of pigs changed toward a leaner pig, and the heat production and tissue insulation of modern pigs are different than those of the pigs used by Bond *et al.* (1959).

The results indicate that although the model still needs improvement in terms of calibration, it generally responded in the right direction, confirming that the concepts in the development of the model were correct. Genetic differences between pig populations makes it difficult to precisely predict growth, composition and carcass value (Schinckel and Lange, 1996). The output of a simulation model does not necessarily reproduce the exact values measured in the field. It is similar to an experiment done at a different time or location, in the sense that the changes in response due to varying factors are expected to be similar, although the constant terms may be different (Whittemore and Fawcett, 1974). In this sense, the model is making good prediction, allowing different treatments to be compared.

Model development is an iterative-learning process, in which accuracy and generalization result from doing and redoing (Pomar *et al.*, 1991a). The development of this model allowed the identification of the segments where more research is needed. For example, a deeper understanding of the mechanisms of compensatory growth may improve predictions under the circumstances in which it occurs. Among other benefits, the model is useful in defining the direction that future research should follow.

Use of the Model

A series of simulation runs was conducted in order to demonstrate how the model can be useful in practical situations. The objective was to evaluate the effect of environmental temperature on body temperature, heat production, feed consumption, growth rate and feed efficiency of pigs of different body weights. The initial scenario was an environment with 80 % relative humidity, air speed of 0.5 m / s, *ad lib.* feeding and little wallowing possible (the maximum fraction of skin wetted by wallowing was set to 0.15). A variation of this scenario was also simulated, with the objective of testing a strategy for the alleviation of heat stress. For simplicity, all temperatures (air, radiant, floor, feed and water) were kept equal and constant on each run. The constant body weight option was used, in order to evaluate the steady state effects of the environment. On each run, four days were simulated; the first two were used to reach steady state conditions and the next two to collect data.

Environmental temperature was varied from 0 °C to 40 °C, in 1 °C intervals, such that 41 simulation runs were used for each data set. Three data sets were collected. The first two were for pigs weighing 20 and 80 kg, at the conditions described above. The

third data set was obtained using 80 kg pigs under modified environmental conditions designed to relieve heat stress. The changes were an increase of air speed from 0.5 to 2 m / s through the use of fans and a modification in the pens that enabled the pigs to wet their skin completely (either a sprinkler system or a pool of water with adequate depth), with the objective of increasing evaporative heat loss at the skin surface.

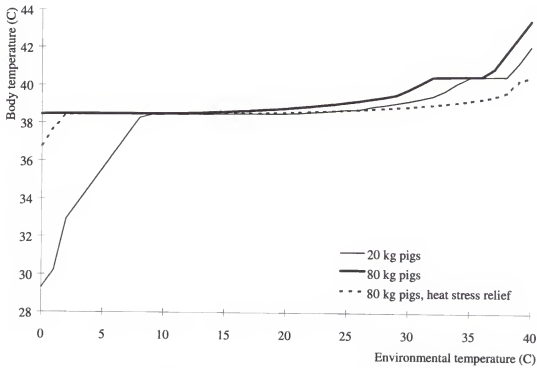


Figure 29. Simulated body temperature in pigs as a function of environmental temperature.

The average rate of heat production is plotted in Figure 29 as a function of environmental temperature. Hypothermia can be observed in this graph as a sharp drop in body temperature below a certain limiting environmental temperature. This limit was 8 °C for the 20 kg pigs and below zero for the 80 kg pigs in the standard (low air speed)

conditions. For the 80 kg pigs in the heat stress relief conditions (high air speed), hypothermia occurred at environmental temperatures below 2 °C. Wallowing is voluntary and under cold conditions the simulated pig will not get itself wet. However, the higher air speed increased convective heat loss and make the 'heat stress relief' pigs more sensitive to cold.

Note that the absolute values of these temperatures depend on the maximum additional heat production rate, which was arbitrarily set. Therefore, the temperature below which hypothermia occurs is not necessarily correct. However, the relation between data sets is useful. Although the values are not exact, it is possible to conclude (as expected) that 20 kg pigs are more sensitive to cold stress and become hypothermic more easily than 80 kg pigs. Furthermore, increasing air speed makes the 80 kg pigs more subject to hypothermia, although the effect is not as significant as the change in body weight.

On the other extreme, 80 kg pigs were more sensitive to heat stress than 20 kg pigs. It is possible to observe a gradual increase in body temperature as environmental temperature rises. At a certain point, body temperature stops rising, due to a decrease in feed intake, which reduces growth and heat production. After the animal completely stops eating, heat production is at the minimum and all the mechanisms for increasing heat loss are at the maximum. If environmental temperature increases further, the animal becomes hyperthermic, which can be observed on the graph as a fast rise in body temperature. For 80 kg pigs, this occurs at environmental temperatures above 36 °C, while for 20 kg pigs it only happens above 38 °C.

The 80 kg pigs under the heat stress relief conditions are able to get wet and also have the benefit of higher air speed to increase evaporative heat loss. For this reason, they can withstand higher temperatures than even the 20 kg pig. These pigs were not hyperthermic even at 40 °C, when they had barely reached the plateau where growth reduction occurs. Although not shown in the graph, hyperthermia would occur in these pigs above 41 °C. This indicates that the use of sprays and fans would be the equivalent of a 5 °C reduction in environmental temperature, from the point of view of preventing hyperthermia.

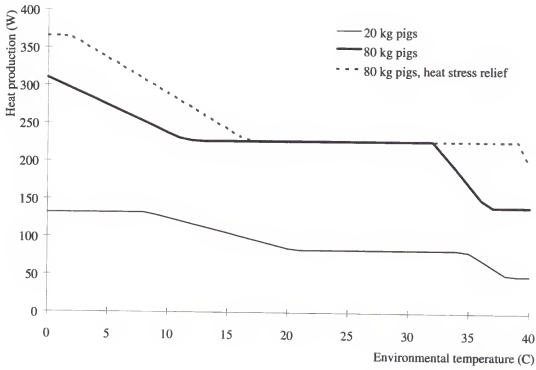


Figure 30. Simulated rate of heat production in pigs as a function of environmental temperature.

The simulated rate of heat production is shown in Figure 30. In all three cases, there was an increase in heat production when body temperature fell below the lower critical temperature of 38.5 °C, which prevented further drops in body temperature as environmental temperature decreased. The temperatures below which hypothermia began correspond to the points where maximum heat production potential was reached. These values were 131.9 W for the 20 kg pigs and 365.7 W for the 80 kg pigs in the heat stress relief environment, which correspond to 13.95 and 13.67 W / $kg^{0.75}$, respectively. The 80 kg pigs in a low air speed environment were not hypothermic in the simulated temperature range, and therefore did not reach their maximum heat production potential.

Within the thermoneutral zone, heat production was stable at 82.6 W for the 20 kg pigs and 226.2 W for the 80 kg pigs, which correspond to 8.73 and 8.46 W / $kg^{0.75}$, respectively. Although the heat production rate of 80 kg pigs is almost triple that of 20 kg pigs, their value is similar in terms of metabolic weight. The drop in heat production under high environmental temperatures corresponds to the range of temperatures in Figure 29 where body temperature was stable. This reduction in heat production occurred due to a reduction in the rate of weight gain. When the animals decreased weight gain, the heat production associated with the inefficiencies in growth was also reduced.

When the pigs were hyperthermic, feed intake was zero and the animals lost weight. At this point, heat production was minimum, and corresponded to basal metabolism with no growth. This minimum was 49.5 W for 20 kg pigs and 139.9 W for 80 kg pigs, which is equal to 5.23 W / $kg^{0.75}$ in both cases.

The use of fans and a system to allow wallowing increased the temperature limits between which heat production was stable. The increase was greater on the high end of

the scale, which made the thermoneutral zone wider for the 'heat stress relief' pigs. Essentially, this means that those pigs were comfortable in a wider range of temperatures. Since the pigs could choose whether or not to get themselves wet, they did so at higher temperatures but not at lower ones. This caused the high end of the thermoneutral zone to be shifted upward, while the low end was unaffected. The increase in air speed, however, did not depend on the pigs' behavior, and it shifted both the low and high end of the thermoneutral zone upward. Of course, cold stress could easily be avoided in a practical situation by simply turning off the fans when ambient temperature is low.

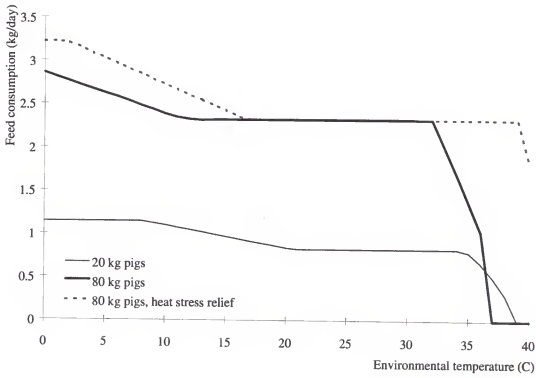


Figure 31. Simulated feed consumption in pigs as a function of environmental temperature.

Simulated feed consumption is shown in Figure 31. Thermoneutral feed intake was 0.821 kg / day for the 20 kg pigs and 2.317 kg / day for the 80 kg pigs. Under cold conditions, feed consumption follows closely the heat production curves, because the animal is eating more to generate heat. Maximum feed intake occurred when the animal was hypothermic, and the maximum values were 1.144 kg / day for 20 kg pigs and 3.221 kg / day for 80 kg pigs.

Under hot conditions, feed intake is reduced and it falls to zero when heat production falls to a minimum value, corresponding to the maintenance energy requirements. The 80 kg pigs begin to reduce feed intake at a lower temperature than the 20 kg pigs. However, the 'heat stress relief' 80 kg pigs were able to withstand a higher environmental temperature than the 20 kg pigs without before reducing feed intake.

The simulated growth rates for different temperatures are shown in Figure 32. The model has no provision for reducing growth or catabolizing tissue under cold stress, so growth rate is maximum in cold conditions, even if the animal is in hypothermia. Under hot conditions, the reduction in growth follows the decrease in feed consumption. When ambient temperature is high enough to cause feed intake to drop to zero, negative growth rates were observed. In these conditions, body tissue was catabolized to meet the animal's maintenance requirements.

Growth rates began to decrease at temperatures above 34 °C for the 20 kg pigs and 32 °C for the 80 kg pigs, again showing that heavier pigs are more sensitive to heat stress. The pigs under the heat stress relief conditions began to show reduction in rate of weight gain at temperatures above 39 °C. This means that the combination of fans and sprinkler corresponded to a 7 °C reduction in temperature, from the point of view of weight gain.

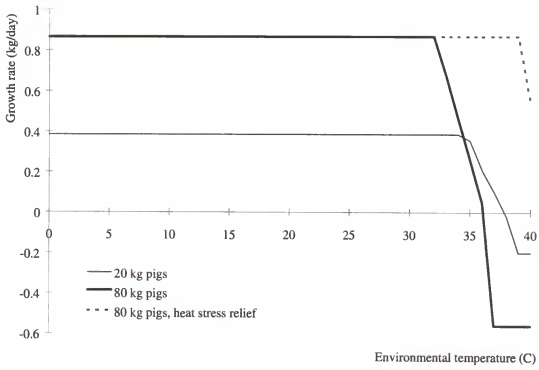


Figure 32. Simulated body growth rate in pigs as a function of environmental temperature.

Feed efficiency (kg of gain per kg of feed consumed) is shown in Figure 33. It was maximum within the thermoneutral zone. As temperature drops into the cold stress zone, feed efficiency decreased due to the greater feed consumption, while the rate of weight gain remained constant. At high temperatures, feed efficiency decreased due to the reduction in weight gain. The same observations concerning maximum tolerable temperatures made for weight gain are valid for feed efficiency. When feed intake dropped to zero, Feed efficiency is meaningless when growth rates are negative (with zero feed intake, growth is negative and calculated feed efficiency is minus infinity).

These examples demonstrate how the model can be used to derive important information about the system being modeled. This was just one example among its

countless uses. More complex situations may be modeled, in which the results are not as intuitive and the model can be very helpful in determining the best strategy to use.

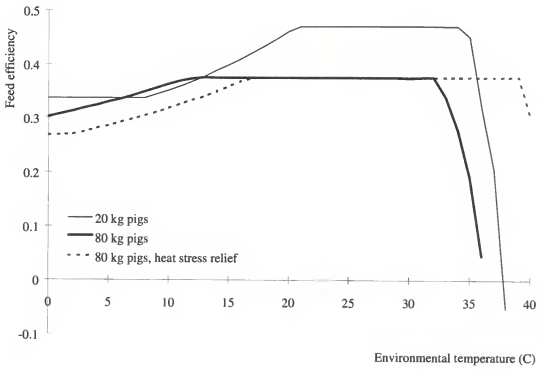


Figure 33. Simulated feed efficiency in pigs as a function of environmental temperature.

Possible Improvements

Modeling is a learning process in which both success and failure lead to a better understanding of the system. Although it is generating good results, there is still much work to be done in terms of refining the model, conducting more field experiments to calibrate parameters, and maybe revising some concepts. It is unlikely that a model of this complexity would be able to correctly predict all the parameters related to swine growth on the first attempt.

As a model is refined, more knowledge is gained about the system and new changes can be proposed, which lead to other changes. At some point, a decision must be made with respect to when the model is good enough. However, it is likely that even at that point many aspects can still be improved greatly.

This simulation model is no different. Originally, essential fat and lean tissue were grouped together as lean growth, and excess fat was grown separately. However, it was observed that in case of protein deficiency, essential fat would not be deposited, which is not necessarily true. Although essential fat is needed for protein growth, protein is not needed (at least as defined in the model) for fat growth. The correction was to combine essential and excess fat growth, separating essential fat from lean tissue. At the same time that this was done, water, which was considered a part of lean tissue, was separated, in order to facilitate further improvements in the model, when the relation between body water and lean tissue may be assumed not to be constant. In order to guide future studies, some suggestions are made about the parts of the model which would most likely gain from being developed in more detail.

The feed digestion process described in the model, although practical from a mathematical point of view, does not reflect how digestion really works in the pig. More detailed models of digestion were developed (Usry *et al.*, 1991; Bridges *et al.*, 1992a; Bastianelli *et al.*, 1996), and feed intake was also studied and modeled in more detail (McDonald *et al.*, 1991; Nienaber *et al.*, 1990a, 1991, 1996; McDonald and Nienaber, 1994; Xin and DeShazer, 1992). A possible improvement of the model would be to refine the feed intake and digestion processes, so that they would reflect better the physiology of

the pig. The definition of different compartments for the different portions of the digestive tract would most likely improve the model's behavior.

A digestion matrix can still be used, but the determination of the coefficients of that matrix may be conducted in a more elaborate scheme than simply assuming constants. For example, the presence of certain substances in the feed can alter the digestibility of some nutrients. To account for that, a provision can be made to change the digestion matrix when such substances are present. The digestion matrix can be defined as a function of feed composition, which would make the transformation a non-linear one, since the matrix coefficients would be affected by the vector of ingested nutrients.

Another improvement would be to subdivide the vector of nutrient net energy into separate vectors for each production function (maintenance, lean tissue deposition and excess fat tissue deposition). Currently, the same vector is used for all functions, and this does not reflect differences in efficiency. For example, the use of fat to form fat tissue is more efficient than the use of carbohydrates, but this may not be true when forming protein tissue.

As mentioned before, the nutrient vectors (nutrient pool and ingested nutrients) can be expanded, to include amino acids, minerals, vitamins, essential fatty acids and any other substances which may seem useful to better simulate metabolism.

Currently, tissue growth is affected only when the nutrients in the nutrient pool are exhausted, at which point the animal immediately shifts from depositing tissue to possibly catabolizing tissue. A more realistic approach would be to have the rate of tissue deposition gradually decrease as the amount of energy in the nutrient pool is reduced,

until it finally reached zero when the nutrients in the nutrient pool were zero or close to zero.

Although there is still much work to be done, the model is in many ways behaving like a pig. The objective of developing the framework of a model was fulfilled. Even if not all portions of the model are sufficiently accurate, this frame is a major fundamental step, necessary for future modeling work. The model is based strongly on physical principles, which gives it a good theoretical foundation on which future refinements can be based.

CHAPTER 12
CONCLUSION

Growth and development of swine was successfully modeled and simulated, within the standards proposed in this work. Although this was only the first step in developing a comprehensive model, the simulated results already show a close resemblance to the way pigs react to different conditions. In particular, it was demonstrated how environmental temperature affects body temperature, heat production, feed intake, growth rate and feed efficiency in pigs of different sizes. The effect of a simple method for alleviating heat stress on each of the variables was also simulated, and it was shown how this method would be as effective as a reduction in ambient temperature.

The general model integrates the different modules and each module functions with nearly complete independence of the others. The model can be refined and expanded easily, without having to rewrite code for the modules that do not change. The use of an object-oriented programming language permitted the separation of modules into independent units. For example, the Climate module, which was not implemented, can be easily incorporated into the model, by altering only the Microenvironment module to accept external data. All other modules could remain unchanged and the simulation program would function as it did before.

In the process of developing the model, specific areas where more research is needed were identified. In particular, the genetic improvement of the worldwide pig

population to leaner pigs made some of the coefficients that were determined in the past obsolete. New research is needed to determine growth curves, heat production rates and nutrient requirements for each productive function, for the current pig genotypes. Furthermore, whatever new results become available now will probably have to be revised again in the future, because the genotypes will continue to evolve.

The simulation model was implemented as a software package called 'SwineSim', which runs on personal computers under a commonly used operating system. SwineSim contains a graphical user interface that makes it simple to use. The user can change practically any of the simulation parameters and observe the results in the simulated output. SwineSim allows the user to visualize a series of graphs containing the results of the simulation, and it also generates two different types of output files, containing simulated data, which can be easily imported into a spreadsheet for further analysis.

Besides regular simulation output, SwineSim also incorporates an expert system that analyzes the simulation results and provides the user with recommendations of possible improvements in the production system. The expert system interface was smoothly integrated with the simulation model, giving it an advantage over the other existing models in terms of usefulness. The expert system allows less knowledgeable producers to benefit from the use of the SwineSim package without having to interpret the results of the simulation.

The flexible knowledge base allows new knowledge to be easily incorporated into the expert system. The SwineSim software program makes it very simple to change the coefficients of the model and observe the results of new simulations. The combination of

a simulation model and an expert system makes this a powerful management tool that can be used to help optimize swine production systems in the future.

APPENDIX A
DERIVATION OF FORMULAS AND COEFFICIENTS

Volume of Feed in the Digestive Tract

The original differential equation is:

$$\frac{dV_i(t)}{dt} = r_i \cdot \left(V_{max} - V_i(t)\right) - r_d \cdot V_i(t) \tag{157}$$

Rearranging the terms:

$$\frac{dV_i(t)}{r_i \cdot V_{max} - r_i \cdot V_i(t) - r_d \cdot V_i(t)} = dt \tag{158}$$

$$\frac{dV_i(t)}{V_i(t) - \frac{r_i}{r_i + r_d} \cdot V_{max}} = -\left(r_i + r_d\right) \cdot dt \tag{159}$$

Integrating and adding the constant term:

$$\int \frac{dV_i(t)}{V_i(t) - \frac{r_i}{r_i + r_d} \cdot V_{max}} = C - \left(r_i + r_d\right) \cdot \int dt \tag{160}$$

Solving the integral:

$$\ln\left(V_i(t) - \frac{r_i}{r_i + r_d} \cdot V_{max}\right) = C - \left(r_i + r_d\right) \cdot t \tag{161}$$

$$V_i(t) - \frac{r_i}{r_i + r_d} \cdot V_{max} = c \cdot e^{-(r_i + r_d) \cdot t} \tag{162}$$

For t = 0:

$$V_i(0) - \frac{r_i}{r_i + r_d} \cdot V_{max} = c \tag{163}$$

Therefore:

$$V_i(t) - \frac{r_i}{r_i + r_d} \cdot V_{max} = \left(V_i(0) - \frac{r_i}{r_i + r_d} \cdot V_{max} \right) \cdot e^{-(r_i + r_d)t} \tag{164}$$

$$V_i(t) = V_i(0) \cdot e^{-(r_i + r_d)t} + \frac{r_i}{r_i + r_d} \cdot V_{max} \cdot \left(1 - e^{-(r_i + r_d)t}\right) \tag{165}$$

Feed Intake Over a Time Step

The original integral is:

$$R_i = \frac{1}{v_f} \cdot \int_0^{\Delta t} r_i \cdot \left(V_{max} - V_i(t)\right) \cdot dt \tag{166}$$

Substituting $V_i(t)$:

$$R_i = \frac{1}{v_f} \cdot \int_0^{\Delta t} r_i \cdot \left(V_{max} - \left(V_i \cdot e^{-(r_i + r_d)t} + \frac{r_i}{r_i + r_d} \cdot V_{max} \cdot \left(1 - e^{-(r_i + r_d)t}\right) \right) \right) \cdot dt \tag{167}$$

Rearranging the terms:

$$R_i = \frac{1}{v_f} \cdot \frac{r_i}{r_i + r_d} \cdot \int_0^{\Delta t} \left(r_d \cdot V_{max} + \left(r_i + r_d\right) \cdot \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot e^{-(r_i + r_d)t} \right) \cdot dt \tag{168}$$

Solving the integral:

$$R_i = \frac{1}{v_f} \cdot \frac{r_i}{r_i + r_d} \cdot \left(r_d \cdot V_{max} \cdot t - \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot e^{-(r_i + r_d)t} \right) \Bigg|_0^{\Delta t} \tag{169}$$

Substituting the limits:

$$R_i = \frac{1}{v_f} \cdot \frac{r_i}{r_i + r_d} \cdot \left(r_d \cdot V_{max} \cdot (\Delta t - 0) - \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot \left(e^{-(r_i + r_d)\Delta t} - 1 \right) \right) \quad (170)$$

Rearranging the terms:

$$R_i = \frac{1}{v_f} \cdot \frac{r_i}{r_i + r_d} \cdot \left(r_d \cdot V_{max} \cdot \Delta t + \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot \left(1 - e^{-(r_i + r_d)\Delta t} \right) \right) \quad (171)$$

Feed Digestion Over a Time Step

The original integral is:

$$R_d = \frac{1}{V_i(0) + R_i \cdot v_f} \cdot \int_0^{\Delta t} r_d \cdot V_i(t) \cdot dt \quad (172)$$

Substituting $V_i(t)$:

$$R_d = \frac{1}{V_i + R_i \cdot v_f} \cdot \int_0^{\Delta t} r_d \cdot \left(V_i \cdot e^{-(r_i + r_d)t} + \frac{r_i}{r_i + r_d} \cdot V_{max} \cdot \left(1 - e^{-(r_i + r_d)t} \right) \right) \cdot dt \quad (173)$$

Rearranging the terms:

$$R_d = \frac{1}{V_i + R_i \cdot v_f} \cdot \frac{r_d}{r_i + r_d} \cdot \int_0^{\Delta t} \left(r_i \cdot V_{max} - (r_i + r_d) \cdot \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot e^{-(r_i + r_d)t} \right) \cdot dt \quad (174)$$

Solving the integral:

$$R_d = \frac{1}{V_i + R_i \cdot v_f} \cdot \frac{r_d}{r_i + r_d} \cdot \left(r_i \cdot V_{max} \cdot t + \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot e^{-(r_i + r_d)t} \right) \Bigg|_0^{\Delta t} \quad (175)$$

Substituting the limits:

$$R_d = \frac{1}{V_i + R_i \cdot v_f} \cdot \frac{r_d}{r_i + r_d} \cdot \left(r_i \cdot V_{max} \cdot (\Delta t - 0) + \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot \left(e^{-(r_i + r_d)\Delta t} - 1 \right) \right) \quad (176)$$

Rearranging the terms:

$$R_d = \frac{1}{V_i + R_i \cdot v_f} \cdot \frac{r_d}{r_i + r_d} \cdot \left(r_i \cdot V_{max} \cdot \Delta t - \left(\frac{r_i}{r_i + r_d} \cdot V_{max} - V_i \right) \cdot \left(1 - e^{-(r_i + r_d)\Delta t} \right) \right) \quad (177)$$

Thermal Conductance of the Floor

Using the method of Bruce and Clark (1979) :

$$C_f = \frac{1}{R_{f45} \cdot \left(\frac{M}{45} \right)^{0.33} \cdot \left(\frac{A_f}{0.2 \cdot A} \right) \cdot N^{0.5}} = \frac{45^{\frac{1}{3}} \cdot 0.2}{R_{f45} \cdot W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \quad (178)$$

For concrete slats ($R_{f45} = 0.07$ °C · m^2 / W):

$$C_f = \frac{10.16}{W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \quad (179)$$

For wooden slats ($R_{f45} = 0.23$ °C · m^2 / W):

$$C_f = \frac{3.09}{W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \quad (180)$$

For straw-bedded floors ($R_{f45} = 0.5$ °C · m^2 / W):

$$C_f = \frac{1.42}{W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \quad (181)$$

For insulated asphalt floors ($R_{f45} = 0.4$ °C · m^2 / W):

$$C_f = \frac{1.78}{W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \quad (182)$$

For metal mesh floors ($R_f = R_a$; $R_{f45} = 0.12$ °C · m^2 / W):

$$C_f = \frac{5.93}{W_b^{\frac{1}{3}} \cdot k_f \cdot N^{0.5}} \quad (183)$$

Convective Vapor Transfer Coefficient

Using the Lewis relation, as described in the text:

$$h_E = \frac{h_C}{c_{pa}} \cdot \left(\frac{W_a}{p_{wa}} \right)_{20°C} \tag{184}$$

Substituting the coefficients for air at 20 °C:

$$h_E = \frac{0.014758}{0.2401 \frac{cal}{g \cdot K} \cdot 4.1868 \frac{J}{cal} \cdot 2338.9 Pa} \cdot h_C \tag{185}$$

Solving:

$$h_E = 6.28 \cdot 10^{-6} \frac{g \cdot °C}{J \cdot Pa} \cdot h_C \tag{186}$$

Fractions of Skin Wetted by Perspiration

For a 50 kg pig and 0.15 m / s air velocity:

$$h_E = 6.28 \cdot 10^{-6} \frac{g \cdot °C}{J \cdot Pa} \cdot 15.7 \frac{W \, kg^{0.13} \, s^{0.6}}{m^{2.6} \cdot °C} \cdot \frac{\left(0.15 \frac{m}{s}\right)^{0.6}}{(50 kg)^{0.13}} = 19 \cdot 10^{-6} \frac{g}{s \cdot m^2 \cdot Pa} \tag{187}$$

For skin temperature at 32 °C, saturation vapor pressure is:

$$p_{ws0} = 1 \cdot 37.729 \text{ mm Hg} \cdot 133.3224 \frac{Pa}{mm \, Hg} = 5030 \text{ Pa} \tag{188}$$

For air at 18 °C and 50 % relative humidity, vapor pressure is:

$$p_{wa0} = 0.5 \cdot 15.477 \text{ mm Hg} \cdot 133.3224 \frac{Pa}{mm \, Hg} = 1032 \text{ Pa} \tag{189}$$

Using these values for the minimum water diffusion rate of 10 $g / m^2 \cdot h$, the minimum fraction of wetted skin is:

$$r_{wp0} = \frac{10 \frac{g}{m^2 \cdot h} \cdot \frac{1h}{3600s}}{h_E \cdot \left(p_{ws0} - p_{wa0} \right)} = \frac{\frac{1}{360} \frac{g}{m^2 \cdot s}}{19 \cdot 10^{-6} \frac{g}{s \cdot m^2 \cdot Pa} \cdot (5030 Pa - 1032 Pa)} = 0.037 \tag{190}$$

For skin temperature at 35 °C, saturation vapor pressure is:

$$p_{ws1} = 1 \cdot 42.175 \text{ mm Hg} \cdot 133.3224 \frac{\text{Pa}}{\text{mm Hg}} = 5623 \text{ Pa} \tag{191}$$

For air at 30 °C and 50 % relative humidity, vapor pressure is:

$$p_{wa1} = 0.5 \cdot 31.824 \text{ mm Hg} \cdot 133.3224 \frac{\text{Pa}}{\text{mm Hg}} = 2121 \text{ Pa} \tag{192}$$

Using these values for the maximum water diffusion rate of 30 $g / m^2 \cdot h$, the maximum fraction of wetted skin is:

$$r_{wp1} = \frac{30 \frac{g}{m^2 \cdot h} \cdot \frac{1 h}{3600 s}}{h_E \cdot (p_{ws1} - p_{wa1})} = \frac{\frac{1}{360} \frac{g}{m^2 \cdot s}}{19 \cdot 10^{-6} \frac{g}{s \cdot m^2 \cdot Pa} \cdot (5623 \text{ Pa} - 2121 \text{ Pa})} = 0.042 \tag{193}$$

Using the data of Beckett (1965), saturation vapor pressure for skin at 39 °C is 6999 Pa, and vapor pressure of air at 37 °C dry bulb and 27 °C wet bulb temperatures is 2927 Pa. For a rate of water loss of 95 $g / m^2 \cdot h$, this results in a maximum fraction of wetted skin of:

$$r_{wp1} = \frac{95 \frac{g}{m^2 \cdot h} \cdot \frac{1 h}{3600 s}}{h_E \cdot (p_{ws1} - p_{wa1})} = \frac{\frac{44}{3600} \frac{g}{m^2 \cdot s}}{19 \cdot 10^{-6} \frac{g}{s \cdot m^2 \cdot Pa} \cdot (6999 \text{ Pa} - 2927 \text{ Pa})} = 0.158 \tag{194}$$

Coefficient of Cooling of Exhaled Air

The humidity ratio in the exhaled air (W_x) can be estimated by summing the moisture in the inhaled air to the moisture lost per unit of exhaled air:

$$W_x = W_s(T_a) \cdot \phi_a + \frac{ML}{BR} \cdot v_a(T_a) \tag{195}$$

where $W_s(T_a)$ is the saturation humidity ratio at air temperature T_a, ϕ_a is the relative humidity of air, ML is the rate of moisture loss through the respiratory tract (g / min), BR is the breathing rate (L / min) and $v_a(T_a)$ is the specific volume of air (L / kg) at T_a. The

temperature at which air with humidity ratio W_x is saturated is given by $T_s(W_x)$. The coefficient of cooling of exhaled air (k_x) can be estimated by dividing the difference between this temperature and body core temperature (T_b) by the difference between air and body temperature:

$$k_x = \frac{T_s(W_x) - T_b}{T_a - T_b} \tag{196}$$

Using the data of Morrison *et al.* (1967), k_x can be estimated for five animals, assuming body temperature of 39 °C.

Animal 1 (air at 15.6 °C and 70 % relative humidity, ML = 0.28 g / min, BR = 12 L / min:

$$W_x = W_s(15.6) \cdot 0.7 + \frac{0.28}{12} \cdot v_a(15.6) = 11.093 \cdot 0.7 + \frac{0.28}{12} \cdot 817.6 = 26.842 \tag{197}$$

$$k_x = \frac{T_s(26.842) - T_b}{T_a - T_b} = \frac{29.7 - 39}{15.6 - 39} = 0.397 \tag{198}$$

Animal 2 (air at 29.4 °C and 30 % relative humidity, ML = 0.87 g / min, BR = 33 L / min:

$$W_x = W_s(29.4) \cdot 0.3 + \frac{0.87}{33} \cdot v_a(29.4) = 26.443 \cdot 0.3 + \frac{0.87}{33} \cdot 857.0 = 30.527 \tag{199}$$

$$k_x = \frac{T_s(30.527) - T_b}{T_a - T_b} = \frac{31.9 - 39}{29.4 - 39} = 0.740 \tag{200}$$

Animal 3 (air at 29.4 °C and 50 % relative humidity, ML = 1.12 g / min, BR = 53 L / min:

$$W_x = W_s(29.4) \cdot 0.5 + \frac{1.12}{53} \cdot v_a(29.4) = 26.443 \cdot 0.5 + \frac{1.12}{53} \cdot 857.0 = 31.332 \tag{201}$$

$$k_x = \frac{T_s(31.332) - T_b}{T_a - T_b} = \frac{32.3 - 39}{29.4 - 39} = 0.698 \tag{202}$$

Animal 4 (air at 29.4 °C and 70 % relative humidity, ML = 0.78 g / min, BR = 60 L / min:

$$W_x = W_s(29.4) \cdot 0.7 + \frac{0.78}{60} \cdot v_a(29.4) = 26.443 \cdot 0.7 + \frac{0.78}{60} \cdot 857.0 = 29.651 \tag{203}$$

$$k_x = \frac{T_s(29.651) - T_b}{T_a - T_b} = \frac{31.4 - 39}{29.4 - 39} = 0.792 \tag{204}$$

Animal 5 (air at 29.4 °C and 90 % relative humidity, ML = 0.41 g / min, BR = 47 L / min:

$$W_x = W_s(29.4) \cdot 0.9 + \frac{0.41}{47} \cdot v_a(29.4) = 26.443 \cdot 0.9 + \frac{0.41}{47} \cdot 857.0 = 31.275 \tag{205}$$

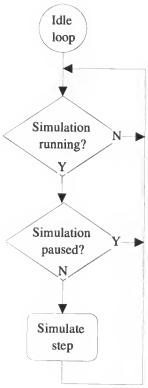
$$k_x = \frac{T_s(31.275) - T_b}{T_a - T_b} = \frac{32.3 - 39}{29.4 - 39} = 0.698 \tag{206}$$

The average value for k_x is:

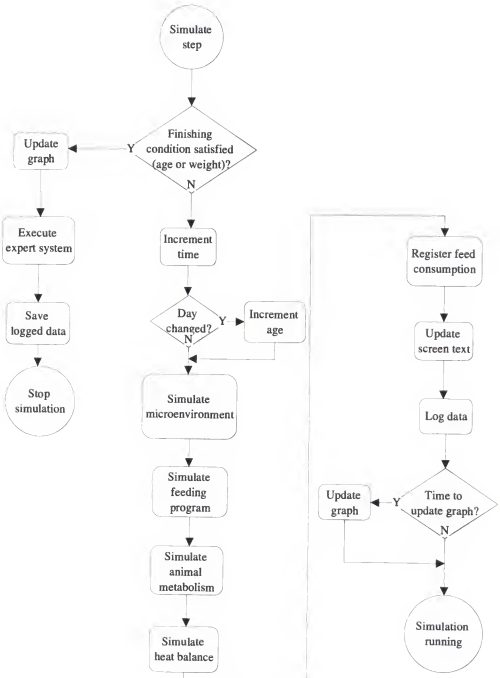
$$k_x = \frac{0.397 + 0.740 + 0.698 + 0.792 + 0.698}{5} = 0.665 \tag{207}$$

APPENDIX B
FLOW CHARTS FOR SWINESIM

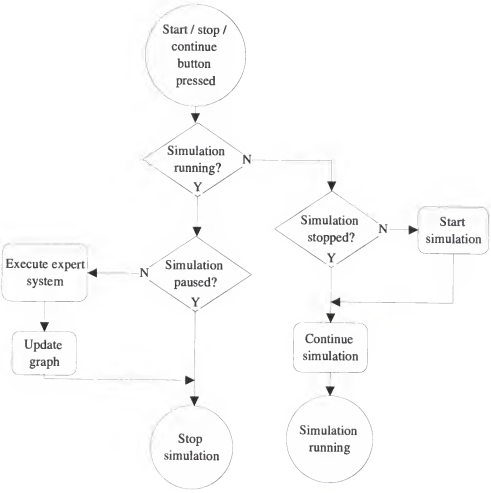
Idle Loop



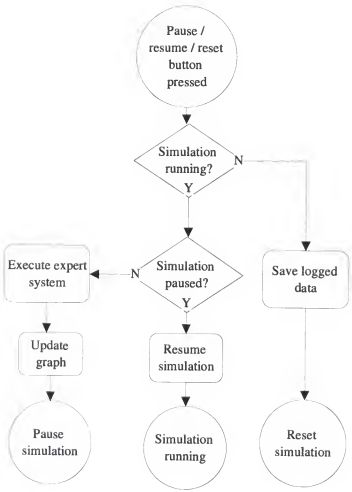
Simulation Step



Start Button Event



Pause Button Event



APPENDIX C
SOURCE CODE FOR SWINESIM

SwineSim.mak

```
# ---------------------------------------------------------------------------
VERSION = BCB.01
# ---------------------------------------------------------------------------
!ifndef BCB
BCB = $(MAKEDIR)\..
!endif
# ---------------------------------------------------------------------------
PROJECT = SwineSim.exe
OBJFILES = SwineSim.obj SwineSimMain.obj SwineSimAboutBox.obj Matrix.obj SwineSimDataTransfer.obj \
    SwineSimSimulation.obj SwineSimEnvironment.obj SwineSimFeedProgram.obj SwineSimHeatBalance.obj \
    SwineSimMetabolism.obj SwineSimExpert.obj SwineSimKnowledgeBase.obj
RESFILES = SwineSim.res
RESDEPEN = $(RESFILES) SwineSimMain.dfm SwineSimAboutBox.dfm
LIBFILES =
DEFFILE =
# ---------------------------------------------------------------------------
CFLAG1 = -Od -Hc -w -k -r- -y -v -vi- -c -a4 -b- -w-par -w-inl -Vx -Ve -x
CFLAG2 = -I.;e:\projects;$(BCB)\include;$(BCB)\include\vcl -H=$(BCB)\lib\vcld.csm
PFLAGS = -AWinTypes=Windows;WinProcs=Windows;DbiTypes=BDE;DbiProcs=BDE;DbiErrs=BDE \
    -U.;e:\projects;$(BCB)\lib\obj;$(BCB)\lib -I.;e:\projects;$(BCB)\include;$(BCB)\include\vcl  -v  -$Y
\
    -$W -$O-  -JPHNV -M
RFLAGS = -i.;e:\projects;$(BCB)\include;$(BCB)\include\vcl
LFLAGS = -L.;e:\projects;$(BCB)\lib\obj;$(BCB)\lib -aa -Tpe -x -v -V4.0
IFLAGS =
LINKER = ilink32
# ---------------------------------------------------------------------------
ALLOBJ = c0w32.obj $(OBJFILES)
ALLRES = $(RESFILES)
ALLLIB = $(LIBFILES) vcl.lib import32.lib cp32mt.lib
# ---------------------------------------------------------------------------
.autodepend

$(PROJECT): $(OBJFILES) $(RESDEPEN) $(DEFFILE)
    $(BCB)\BIN\$(LINKER) @&&!
    $(LFLAGS) +
    $(ALLOBJ), +
    $(PROJECT),, +
    $(ALLLIB), +
    $(DEFFILE), +
    $(ALLRES)
!

.pas.hpp:
    $(BCB)\BIN\dcc32 $(PFLAGS) { $** }

.pas.obj:
    $(BCB)\BIN\dcc32 $(PFLAGS) { $** }

.cpp.obj:
    $(BCB)\BIN\bcc32 $(CFLAG1) $(CFLAG2) -o$* $*

.c.obj:
    $(BCB)\BIN\bcc32 $(CFLAG1) $(CFLAG2) -o$* $**

.rc.res:
    $(BCB)\BIN\brcc32 $(RFLAGS) $<
#---------------------------------------------------------------------------
```

SwineSim.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimH
#define SwineSimH
//---------------------------------------------------------------------------
// Constant defining the fly-shaped cursor:
#define crFlyCursor ((Controls::TCursor) -21)
// Size of names used in the program (including final 0):
#define NAME_SIZE 32
// Number of time steps recorded (and graphed) per day:
#define STEPS_PER_DAY 240
// Maximum number of days to record data:
#define MAX_RECORDED_DAYS 240
// Round-off error (should be very small):
#define ROUND_OFF 0.00000001
// Number of points in the environment graph:
#define ENVGRAPH_DATAPOINTS 240
// Types of actions allowed in the feeding schedule:
enum TActionType {
    actionDiscard,
    actionRemove,
    actionRestore,
    actionLevel,
    actionSupply
};
// Maximum number of daily feeding schedule actions:
#define MAX_ACTIONS 10
// Maximum number of feeding phases:
#define MAX_PHASES 10
// Maximum weight displayed on feeding phase graph (kg):
#define MAX_WEIGHT 120
// Maximum age displayed on feeding phase graph (days):
#define MAX_AGE 200
// Amount of available feed when feeding is ad lib:
#define LOTS_OF_FEED 1
// Nutrients in the feed composition vector:
enum {
    FEED_PROTEIN,
    FEED_FAT,
    FEED_CARBS,
    FEED_FIBER,
    FEED_VOLUME,
    FEED_NUTRIENTS
};
// Number of nutrients in the nutrient pool vector:
enum {
    POOL_PROTEIN,
    POOL_FAT,
    POOL_CARBS,
    POOL_NUTRIENTS
};
//---------------------------------------------------------------------------
// Classes defined in the program:
class TMainWindow;
class TAboutBox;
class Vector;
class Matrix;
struct TVariableDescription;
class TDataTransfer;
struct TSimulatedData;
struct TDailyTotals;
class TSimulation;
class TEnvironment;
struct TAction;
struct TPhase;
struct TDiet;
class TFeedingProgram;
class THeatBalance;
class TMetabolism;
//---------------------------------------------------------------------------
#include "Matrix.h"
#include "SwineSimExpert.h"
#include "SwineSimDataTransfer.h"
#include "SwineSimSimulation.h"
#include "SwineSimEnvironment.h"
#include "SwineSimFeedProgram.h"
#include "SwineSimHeatBalance.h"
#include "SwineSimMetabolism.h"
#include "SwineSimMain.h"
#include "SwineSimAboutBox.h"
//---------------------------------------------------------------------------
#endif
```

SwineSim.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "SwineSim.h"
//---------------------------------------------------------------------------
USERES("SwineSim.res");
USEFORM("SwineSimMain.cpp", MainWindow);
USEFORM("SwineSimAboutBox.cpp", AboutBox);
USEUNIT("Matrix.cpp");
USEUNIT("SwineSimDataTransfer.cpp");
USEUNIT("SwineSimSimulation.cpp");
USEUNIT("SwineSimEnvironment.cpp");
USEUNIT("SwineSimFeedProgram.cpp");
USEUNIT("SwineSimHeatBalance.cpp");
USEUNIT("SwineSimMetabolism.cpp");
USEUNIT("SwineSimExpert.cpp");
USEUNIT("SwineSimKnowledgeBase.cpp");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->Title = "Swine Simulator 1.0";
        Screen->Cursors[crFlyCursor] = LoadCursorFromFile("Fly.cur");
        Application->CreateForm(__classid(TMainWindow), &MainWindow);
        Application->CreateForm(__classid(TAboutBox), &AboutBox);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//---------------------------------------------------------------------------
```

SwineSim.res

Binary file containing the resources used by the application.

SwineSimAboutBox.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimAboutBoxH
#define SwineSimAboutBoxH
//---------------------------------------------------------------------------
#include <vcl\System.hpp>
#include <vcl\Windows.hpp>
#include <vcl\SysUtils.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Graphics.hpp>
#include <vcl\Forms.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Buttons.hpp>
#include <vcl\ExtCtrls.hpp>
#include <vcl\MPlayer.hpp>
//---------------------------------------------------------------------------
class TAboutBox : public TForm
{
__published:
    TPanel *Panel1;
    TImage *PigImage;
    TLabel *ProductName;
```

```
    TLabel *Version;
    TLabel *Copyright;
    TLabel *Embrapa;
    TButton *OKButton;
    TLabel *Author;
    TLabel *UF;
    TLabel *CNPq;
    TMediaPlayer *PigSound;
    TLabel *PigLabel;
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall PigImageClick(TObject *Sender);
    void __fastcall FormShow(TObject *Sender);
private:
public:
    virtual __fastcall TAboutBox(TComponent* AOwner);
};
//---------------------------------------------------------------------------
extern TAboutBox *AboutBox;
//---------------------------------------------------------------------------
#endif
```

SwineSimAboutBox.cpp

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "SwineSim.h"
#include "SwineSimAboutBox.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TAboutBox *AboutBox;
//---------------------------------------------------------------------
__fastcall TAboutBox::TAboutBox(TComponent* AOwner)
    : TForm(AOwner)
{
}
//---------------------------------------------------------------------
void __fastcall TAboutBox::FormCreate(TObject *Sender)
{
    int i;
    Cursor = crFlyCursor;
    for (i = 0; i < ControlCount; i ++)
        Controls[i]->Cursor = crFlyCursor;
    try { PigImage->Picture->LoadFromFile ("PIG.BMP"); }
    catch (...) {}
    try { PigSound->Open (); }
    catch (EMCIDeviceError &E) {}
}
//---------------------------------------------------------------------
void __fastcall TAboutBox::PigImageClick(TObject *Sender)
{
    if (PigSound->Mode == mpPlaying)
        PigSound->Pause ();
    else {
        try { PigSound->Play (); }
        catch (EMCIDeviceError &E) {}
    }
}
//---------------------------------------------------------------------
void __fastcall TAboutBox::FormShow(TObject *Sender)
{
    static bool CoolLabel = true;
    PigLabel->Caption = CoolLabel ? "Pigs are cool!" : "Pigs are hot!";
    CoolLabel = ! CoolLabel;
}
//---------------------------------------------------------------------
```

SwineSimAboutBox.dfm

Binary file containing the layout of the About Box form.

SwineSimMain.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimMainH
#define SwineSimMainH
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\ComCtrls.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\GraphSvr.hpp>
#include <vcl\OleCtrls.hpp>
#include <vcl\Menus.hpp>
#include <vcl\Dialogs.hpp>
#include <vcl\ExtCtrls.hpp>
#include "Grids.hpp"
#include <vcl\Forms.hpp>
#include <vcl\MPlayer.hpp>
#include "SwineSim.h"
//---------------------------------------------------------------------------
class TMainWindow : public TForm
{
    friend TDataTransfer;
__published:  // IDE-managed Components
    TMainMenu *MainMenu;
    TMenuItem *MenuFile;
    TMenuItem *FileNew;
    TMenuItem *FileOpen;
    TMenuItem *FileSave;
    TMenuItem *FileSaveAs;
    TMenuItem *N1;
    TMenuItem *FilePrint;
    TMenuItem *FilePrinterSetup;
    TMenuItem *N2;
    TMenuItem *FileExit;
    TMenuItem *MenuHelp;
    TMenuItem *HelpAbout;
    TOpenDialog *FileOpenDialog;
    TSaveDialog *FileSaveDialog;
    TPrinterSetupDialog *PrinterSetupDialog;
    TPrintDialog *PrintDialog;
    TPageControl *PageControl;
    TTabSheet *EnvTab;
    TLabel *Label1;
    TLabel *Label2;
    TLabel *Label3;
    TLabel *Label4;
    TLabel *Label5;
    TLabel *Label6;
    TLabel *Label7;
    TLabel *Label8;
    TLabel *Label9;
    TLabel *Label10;
    TLabel *Label11;
    TLabel *Label12;
    TLabel *Label13;
    TLabel *Label14;
    TLabel *Label15;
    TLabel *Label16;
    TLabel *Label17;
    TLabel *Label18;
    TLabel *Label19;
    TLabel *Label20;
    TLabel *Label21;
    TLabel *Label22;
    TLabel *Label23;
    TLabel *Label24;
    TLabel *Label25;
    TLabel *Label26;
    TLabel *Label27;
    TLabel *Label28;
    TLabel *Label29;
    TLabel *Label30;
    TLabel *Label31;
    TLabel *Label32;
    TLabel *Label33;
    TLabel *Label34;
    TLabel *Label35;
    TLabel *Label36;
    TLabel *Label37;
    TGraphicsServer *EnvGraph;
    TEdit *Ta_min;
    TEdit *tTa_min;
```

```
TEdit *Ta_max;
TEdit *tTa_max;
TEdit *Tr_min;
TEdit *tTr_min;
TEdit *Tr_max;
TEdit *tTr_max;
TEdit *Tf_min;
TEdit *tTf_min;
TEdit *Tf_max;
TEdit *tTf_max;
TEdit *phia_max;
TEdit *tphia_max;
TEdit *phia_min;
TEdit *tphia_min;
TEdit *vaf;
TEdit *tVa_on;
TEdit *tVa_off;
TEdit *va_min1;
TEdit *tVa_min1;
TEdit *va_max1;
TEdit *tVa_max1;
TEdit *va_min2;
TEdit *tVa_min2;
TEdit *va_max2;
TEdit *tVa_max2;
TButton *EnvBtnRefresh;
TButton *EnvBtnPrint;
TButton *EnvBtnAccept;
TButton *EnvBtnCancel;
TTabSheet *FeedTab;
TLabel *Label138;
TLabel *PhaseTotal;
TLabel *Label139;
TLabel *PhaseCurrent;
TLabel *Label140;
TLabel *Label141;
TLabel *Label142;
TLabel *Label143;
TLabel *Label144;
TLabel *Label145;
TLabel *Label146;
TLabel *Label147;
TLabel *Label148;
TLabel *Label149;
TLabel *Label150;
TLabel *PhaseAction;
TLabel *Label151;
TLabel *Label152;
TLabel *Label153;
TLabel *Label154;
TLabel *Label155;
TRadioGroup *PhaseTransitionType;
TButton *FeedBtnAccept;
TButton *FeedBtnCancel;
TButton *FeedBtnRefresh;
TButton *FeedBtnPrint;
TPanel *PhasePanel;
TEdit *PhaseName;
TComboBox *PhaseDiet;
TCheckBox *PhaseEnabled;
TCheckBox *PhaseAdLib;
TEdit *PhaseStartAge;
TEdit *PhaseStartWeight;
TEdit *PhaseInitialFeed;
TEdit *PhaseWeeklyInc;
TGroupBox *PhaseActionGroup;
TCheckBox *PhaseActionEnabled;
TEdit *PhaseTimeOfAction;
TComboBox *PhaseActionType;
TEdit *PhaseActionAmount;
TUpDown *PhaseUpDownAction;
TScrollBar *PhaseScroll;
TGraphicsServer *PhaseGraph;
TTabSheet *SimTab;
TLabel *Label58;
TLabel *Label59;
TLabel *Label60;
TLabel *Label61;
TLabel *SimDaysLabel;
TLabel *SimKgLabel;
TLabel *Label62;
TLabel *Label63;
TEdit *SimStartAge;
TEdit *SimStartWeight;
TRadioButton *SimRbFinishDays;
TEdit *SimSimulatedDays;
TRadioButton *SimRbFinishWeight;
```

```
TEdit *SimFinishWeight;
TEdit *SimTimeStep;
TRadioGroup *SimGraphType;
TButton *SimBtnStart;
TButton *SimBtnReset;
TButton *SimBtnPrint;
TGraphicsServer *SimGraph;
TTabSheet *HeatTab;
TTabSheet *MetTab;
TStatusBar *StatusBar;
TImage *HiddenImage;
TLabel *Label64;
TEdit *SimTimeOfDay;
TPanel *HiddenPanel;
TLabel *Label65;
TEdit *SimSimulationDay;
TLabel *Label67;
TLabel *Label68;
TLabel *Label69;
TLabel *Label70;
TLabel *Label71;
TEdit *SimCurrentAge;
TEdit *SimCurrentWeight;
TButton *SimBtnRefresh;
TButton *SimBtnAccept;
TButton *SimBtnCancel;
TLabel *Label85;
TEdit *I6;
TEdit *I7;
TEdit *I8;
TEdit *I9;
TEdit *I10;
TEdit *I11;
TEdit *I12;
TLabel *Label123;
TLabel *Label122;
TLabel *Label121;
TLabel *Label120;
TLabel *Label119;
TLabel *Label118;
TLabel *Label86;
TLabel *Label111;
TLabel *Label112;
TLabel *Label113;
TLabel *Label114;
TLabel *Label115;
TLabel *Label116;
TLabel *Label117;
TLabel *Label83;
TLabel *Label124;
TEdit *Pw;
TRadioGroup *EnvFeedTempEqualAir;
TEdit *Ti;
TEdit *N;
TStringGrid *DietGrid;
TLabel *Label147;
TEdit *FeedUseEfficiency;
TGroupBox *FeedDietGroup;
TLabel *Label144;
TLabel *DietTotal;
TLabel *Label146;
TEdit *DietName;
TButton *DietBtnAdd;
TButton *DietBtnInsert;
TButton *DietBtnDelete;
TButton *DietBtnRename;
TLabel *Label143;
TEdit *pa;
TLabel *Label145;
TLabel *Label148;
TStringGrid *DigestionGrid;
TPanel *MetGridTitlePanel;
TLabel *Label170;
TEdit *Emin_;
TEdit *Emax_;
TLabel *Label191;
TEdit *kv;
TLabel *Label193;
TLabel *Label194;
TLabel *Label195;
TLabel *Label196;
TEdit *rL_;
TEdit *rd;
TLabel *Label197;
TLabel *Label198;
TEdit *kE;
TLabel *Label209;
```

```
TGroupBox *MetNutRequirementGroup;
TGroupBox *MetNRMaintenanceGroup;
TGroupBox *MetNRLeanGrowthGroup;
TLabel *Label212;
TEdit *Pmr;
TLabel *Label175;
TLabel *Label190;
TLabel *Label189;
TLabel *Label174;
TLabel *Label188;
TLabel *Label204;
TLabel *Label173;
TLabel *Label203;
TEdit *Emr_;
TEdit *Gmr;
TEdit *Lmr;
TLabel *Label213;
TLabel *Label214;
TLabel *Label202;
TEdit *Pld;
TLabel *Label185;
TEdit *Lld;
TLabel *Label186;
TEdit *Gld;
TLabel *Label187;
TLabel *Label218;
TLabel *Label219;
TLabel *Label220;
TLabel *Label221;
TEdit *Eld;
TLabel *Label208;
TGroupBox *MetNRFatGrowthGroup;
TLabel *Label172;
TLabel *Label184;
TEdit *Efd;
TGroupBox *MetTissueCompGroup;
TLabel *Label168;
TEdit *Plt;
TEdit *Pft;
TLabel *Label169;
TLabel *Label165;
TLabel *Label162;
TEdit *Llt;
TEdit *Lft;
TLabel *Label166;
TLabel *Label163;
TEdit *Glt;
TEdit *Gft;
TLabel *Label167;
TLabel *Label164;
TLabel *Label205;
TLabel *Label206;
TLabel *Label207;
TGroupBox *MetTissueCatGroup;
TLabel *Label222;
TLabel *Label223;
TLabel *Label224;
TLabel *Label225;
TLabel *Label226;
TLabel *Label227;
TLabel *Label228;
TLabel *Label229;
TLabel *Label230;
TLabel *Label231;
TLabel *Label232;
TEdit *Gfc;
TEdit *Glc;
TEdit *Llc;
TEdit *Lfc;
TEdit *Pfc;
TEdit *Plc;
TGroupBox *MetNutEnergyGroup;
TLabel *Label233;
TLabel *Label234;
TLabel *Label235;
TLabel *Label236;
TLabel *Label237;
TLabel *Label238;
TLabel *Label239;
TLabel *Label240;
TLabel *Label241;
TLabel *Label242;
TLabel *Label243;
TEdit *EMP;
TEdit *EML;
TEdit *EMG;
TEdit *ENG;
```

```
TEdit *ENL;
TEdit *ENP;
TLabel *Label151;
TLabel *Label152;
TLabel *Label153;
TLabel *Label159;
TLabel *Label160;
TLabel *Label161;
TLabel *Label171;
TLabel *Label176;
TLabel *Label177;
TLabel *Label178;
TLabel *Label179;
TLabel *Label180;
TLabel *Label181;
TLabel *Label182;
TLabel *Label183;
TLabel *Label201;
TLabel *Label211;
TLabel *Label215;
TLabel *Label216;
TLabel *Label217;
TEdit *kr;
TLabel *Label199;
TLabel *Label200;
TGroupBox *HeatSkinGroup;
TLabel *Label98;
TLabel *Label100;
TLabel *Label101;
TLabel *Label102;
TEdit *rwp0;
TEdit *rwp1;
TEdit *rww0;
TEdit *rww1;
TLabel *Label105;
TLabel *Label104;
TLabel *Label103;
TLabel *Label99;
TLabel *Label131;
TEdit *ka0;
TLabel *Label132;
TEdit *ka1;
TEdit *kf0;
TLabel *Label108;
TLabel *Label109;
TLabel *Label106;
TEdit *kf1;
TEdit *ks0;
TEdit *ks1;
TLabel *Label136;
TLabel *Label135;
TLabel *Label134;
TLabel *Label107;
TLabel *Label133;
TGroupBox *HeatThermalZonesGroup;
TLabel *Label74;
TLabel *Label76;
TEdit *Th;
TLabel *Label75;
TEdit *Te;
TLabel *Label77;
TEdit *Tc;
TLabel *Label81;
TLabel *Label80;
TGroupBox *HeatFloorGroup;
TRadioButton *BuildRbFloorType;
TComboBox *BuildFloorType;
TRadioButton *BuildRbFloorConductance;
TEdit *Cf0;
TLabel *Label130;
TLabel *Label82;
TGroupBox *HeatAnimalGroup;
TLabel *Label88;
TLabel *Label84;
TLabel *Label89;
TLabel *Label90;
TEdit *Ct0_;
TEdit *Ct1_;
TLabel *Label91;
TLabel *Label92;
TLabel *Label128;
TLabel *Label129;
TLabel *Label126;
TLabel *Label127;
TEdit *Ka;
TLabel *Label125;
TGroupBox *HeatRadiationGroup;
```

```
TLabel *Label87;
TEdit *theta;
TEdit *ksa;
TLabel *Label137;
TLabel *Label197;
TGroupBox *HeatVentilationGroup;
TLabel *Label142;
TEdit *kx;
TEdit *rv1_;
TEdit *rv0_;
TLabel *Label141;
TLabel *Label140;
TLabel *Label139;
TLabel *Label166;
TLabel *Label56;
TGroupBox *HeatIngestionGroup;
TLabel *Label138;
TEdit *cpi;
TLabel *Label244;
TLabel *Label245;
TLabel *Label246;
TLabel *Label247;
TLabel *Label248;
TEdit *Rw0;
TEdit *Rw1;
TLabel *Label250;
TLabel *Label252;
TLabel *Label251;
TLabel *Label249;
TGroupBox *HeatEmissivityGroup;
TLabel *Label110;
TEdit *er;
TEdit *el;
TEdit *es;
TLabel *Label95;
TLabel *Label196;
TTabSheet *GrowthTab;
TGroupBox *GrowthGroup2;
TRadioButton *GrowthRb1;
TRadioButton *GrowthRb2;
TGraphicsServer *GrowthGraph;
TTabSheet *ExpertTab;
TLabel *Label266;
TLabel *Label267;
TLabel *Label268;
TLabel *Label269;
TLabel *Label270;
TLabel *Label271;
TLabel *Label272;
TLabel *Label273;
TLabel *Label274;
TLabel *Label275;
TLabel *Label276;
TLabel *Label277;
TLabel *Label278;
TLabel *Label279;
TLabel *Label280;
TLabel *Label281;
TLabel *Label282;
TLabel *Label283;
TLabel *Label284;
TLabel *Label285;
TLabel *Label286;
TLabel *Label287;
TEdit *Growth2_WmaxL;
TEdit *Growth2_aL;
TEdit *Growth2_tmaxL;
TGroupBox *GrowthGroup1;
TLabel *Label253;
TLabel *Label258;
TLabel *Label259;
TLabel *Label260;
TLabel *Label261;
TLabel *Label254;
TLabel *Label94;
TEdit *Growth1_kL;
TEdit *Growth1_kF;
TLabel *Label255;
TLabel *Label256;
TLabel *Label257;
TLabel *Label262;
TLabel *Label263;
TEdit *Growth2_tmaxF;
TEdit *Growth2_aF;
TEdit *Growth2_WmaxF;
TLabel *Label264;
TLabel *Label265;
```

```
TLabel *Label288;
TLabel *Label289;
TRadioButton *GrowthRb3;
TEdit *Growth2_gestation;
TLabel *Label290;
TLabel *Label291;
TButton *GrowthBtnCancel;
TButton *GrowthBtnAccept;
TButton *GrowthBtnPrint;
TButton *GrowthBtnRefresh;
TLabel *Label149;
TEdit *MetFatLeanRatio;
TButton *MetBtnCancel;
TButton *MetBtnAccept;
TButton *HeatBtnCancel;
TButton *HeatBtnAccept;
TCheckBox *Growth2_PhysAge;
TGroupBox *GrowthGroup3;
TLabel *Label295;
TGroupBox *GroupBox2;
TLabel *Label155;
TLabel *Label156;
TEdit *cpl;
TEdit *cpf;
TLabel *Label296;
TEdit *kFL;
TLabel *Label297;
TEdit *kWT;
TLabel *Label298;
TEdit *kEB;
TEdit *kF;
TLabel *Label150;
TLabel *Label157;
TLabel *Label158;
TLabel *Label192;
TEdit *qmax;
TCheckBox *SimConstantWeight;
TShape *HeatCircleShape;
TShape *HeatRedShape;
TMemo *ExpertMemo;
TMemo *CirandaMemo;
TShape *HeatGreenShape;
TShape *HeatBlueShape;
TImage *HiddenFlavio;
void __fastcall FormResize(TObject *Sender);
void __fastcall PageControlChange(TObject *Sender);
void __fastcall FileNewClick(TObject *Sender);
void __fastcall FileOpenClick(TObject *Sender);
void __fastcall FileSaveClick(TObject *Sender);
void __fastcall FileSaveAsClick(TObject *Sender);
void __fastcall FilePrintClick(TObject *Sender);
void __fastcall FilePrinterSetupClick(TObject *Sender);
void __fastcall FileExitClick(TObject *Sender);
void __fastcall Ta_minExit(TObject *Sender);
void __fastcall tTa_minExit(TObject *Sender);
void __fastcall Ta_maxExit(TObject *Sender);
void __fastcall tTa_maxExit(TObject *Sender);
void __fastcall Tr_minExit(TObject *Sender);
void __fastcall tTr_minExit(TObject *Sender);
void __fastcall Tr_maxExit(TObject *Sender);
void __fastcall tTr_maxExit(TObject *Sender);
void __fastcall Tf_minExit(TObject *Sender);
void __fastcall tTf_minExit(TObject *Sender);
void __fastcall Tf_maxExit(TObject *Sender);
void __fastcall tTf_maxExit(TObject *Sender);
void __fastcall phia_maxExit(TObject *Sender);
void __fastcall tphia_maxExit(TObject *Sender);
void __fastcall phia_minExit(TObject *Sender);
void __fastcall tphia_minExit(TObject *Sender);
void __fastcall vafExit(TObject *Sender);
void __fastcall tVa_onExit(TObject *Sender);
void __fastcall tVa_offExit(TObject *Sender);
void __fastcall va_min1Exit(TObject *Sender);
void __fastcall tVa_min1Exit(TObject *Sender);
void __fastcall va_max1Exit(TObject *Sender);
void __fastcall tVa_max1Exit(TObject *Sender);
void __fastcall va_min2Exit(TObject *Sender);
void __fastcall tVa_min2Exit(TObject *Sender);
void __fastcall va_max2Exit(TObject *Sender);
void __fastcall tVa_max2Exit(TObject *Sender);
void __fastcall EnvBtnRefreshClick(TObject *Sender);
void __fastcall EnvBtnPrintClick(TObject *Sender);
void __fastcall EnvBtnAcceptClick(TObject *Sender);
void __fastcall EnvBtnCancelClick(TObject *Sender);
void __fastcall PhaseTransitionTypeClick(TObject *Sender);
void __fastcall FeedBtnRefreshClick(TObject *Sender);
void __fastcall PhaseScrollScroll(TObject *Sender, TScrollCode ScrollCode, int &ScrollPos);
```

```
    void __fastcall PhaseNameExit(TObject *Sender);
    void __fastcall PhaseDietChange(TObject *Sender);
    void __fastcall PhaseEnabledClick(TObject *Sender);
    void __fastcall PhaseAdLibClick(TObject *Sender);
    void __fastcall PhaseStartAgeExit(TObject *Sender);
    void __fastcall PhaseStartWeightExit(TObject *Sender);
    void __fastcall PhaseInitialFeedExit(TObject *Sender);
    void __fastcall PhaseWeeklyIncExit(TObject *Sender);
    void __fastcall PhaseActionEnabledClick(TObject *Sender);
    void __fastcall PhaseTimeOfActionExit(TObject *Sender);
    void __fastcall PhaseActionTypeChange(TObject *Sender);
    void __fastcall PhaseActionAmountExit(TObject *Sender);
    void __fastcall DietBtnInsertClick(TObject *Sender);
    void __fastcall SimBtnStartClick(TObject *Sender);
    void __fastcall SimRbFinishWeightClick(TObject *Sender);
    void __fastcall SimRbFinishDaysClick(TObject *Sender);
    void __fastcall DietGrid_SelectCell(TObject *Sender, long Col, long Row, bool &CanSelect);
    void __fastcall DietBtnAddClick(TObject *Sender);
    void __fastcall DietBtnDeleteClick(TObject *Sender);
    void __fastcall DietBtnRenameClick(TObject *Sender);
    void __fastcall DietName_Exit(TObject *Sender);
    void __fastcall FeedBtnAcceptClick(TObject *Sender);
    void __fastcall FeedBtnCancelClick(TObject *Sender);
    void __fastcall FeedBtnPrintClick(TObject *Sender);
    void __fastcall SimBtnResetClick(TObject *Sender);
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall SimGraphTypeClick(TObject *Sender);
    void __fastcall FeedUseEfficiency_Exit(TObject *Sender);
    void __fastcall SimBtnPrintClick(TObject *Sender);
    void __fastcall SimBtnRefreshClick(TObject *Sender);
    void __fastcall SimBtnAcceptClick(TObject *Sender);
    void __fastcall SimBtnCancelClick(TObject *Sender);
    void __fastcall PhaseUpDownActionClick(TObject *Sender, TUDBtnType Button);
    void __fastcall SimStartAgeExit(TObject *Sender);
    void __fastcall SimStartWeightExit(TObject *Sender);
    void __fastcall SimSimulatedDaysExit(TObject *Sender);
    void __fastcall SimFinishWeightExit(TObject *Sender);
    void __fastcall SimTimeStepExit(TObject *Sender);
    void __fastcall HelpAboutClick(TObject *Sender);
    void __fastcall BuildRbFloorTypeClick(TObject *Sender);
    void __fastcall BuildRbFloorConductanceClick(TObject *Sender);
    void __fastcall BuildFloorTypeChange(TObject *Sender);
    void __fastcall EnvFeedTempEqualAirClick(TObject *Sender);
    void __fastcall GrowthRbClick(TObject *Sender);
    void __fastcall GrowthRb2Click(TObject *Sender);
    void __fastcall HiddenImageClick(TObject *Sender);
    void __fastcall GrowthRbClick(TObject *Sender);
    void __fastcall GrowthBtnRefreshClick(TObject *Sender);
    void __fastcall GrowthBtnPrintClick(TObject *Sender);
    void __fastcall GrowthBtnAcceptClick(TObject *Sender);
    void __fastcall GrowthBtnCancelClick(TObject *Sender);
    void __fastcall HeatBtnAcceptClick(TObject *Sender);
    void __fastcall HeatBtnCancelClick(TObject *Sender);
    void __fastcall MetBtnAcceptClick(TObject *Sender);
    void __fastcall MetBtnCancelClick(TObject *Sender);
    void __fastcall MetFatLeanRatioExit(TObject *Sender);
    void __fastcall CirandaMemoDblClick(TObject *Sender);*
    void __fastcall HeatRedShapeDragOver(TObject *Sender, TObject *Source, int X,
    int Y, TDragState State, bool &Accept);
    void __fastcall HeatGreenShapeDragOver(TObject *Sender, TObject *Source, int X,
    int Y, TDragState State, bool &Accept);
    void __fastcall HeatBlueShapeDragOver(TObject *Sender, TObject *Source, int X,
    int Y, TDragState State, bool &Accept);
    void __fastcall HeatCircleShapeMouseDown(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
    void __fastcall HiddenFlavioClick(TObject *Sender);
    void __fastcall DietGridSetEditText(TObject *Sender, long ACol, long ARow, const AnsiString Value);
private:  // User declarations
    TPhase Phase [MAX_PHASES];
    Vector FeedMetabEnergy;
    void InitEnvTab (void);
    void InitFeedTab (void);
    void InitHeatTab (void);
    void InitMetaTab (void);
    void UpdateEnvGraph (void);
    void UpdatePhaseFields (void);
    void UpdatePhaseGraph (void);
    void UpdateGrowthGraph (void);
    void ValidateValue (TEdit *Edit, double min, double max);
    void ValidateEnvTime (TEdit * TimeEdit, double Tmin, double Tmax);
    void ValidatePhaseAge (int CurPhase);
    void ValidatePhaseWeight (int CurPhase);
    void ValidatePhaseInitialFeed (int CurPhase);
    void ValidatePhaseTimeOfAction (int CurPhase, int CurAction);
    void ValidatePhaseActionAmount (int CurPhase, int CurAction);
    void DietAdd (void);
    void DietInsert (void);
```

```
    void DietDelete (void);
    void DietRename (void);
#define MAX_ACTIONS 10
public:         // User declarations
    bool SimulationRunning;
    bool SimulationStopped;
    __fastcall TMainWindow(TComponent* Owner);
    void __fastcall IdleEvent(System::TObject* Sender, bool &Done);
    void UpdateSimGraph (void);
};
//---------------------------------------------------------------------------
extern TMainWindow *MainWindow;
//---------------------------------------------------------------------------
#endif
```

SwineSimMain.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include <io.h>
#include "Printers.hpp"
#include "SwineSim.h"
#include "SwineSimMain.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TMainWindow *MainWindow;
//---------------------------------------------------------------------------
__fastcall TMainWindow::TMainWindow(TComponent* Owner)
    : TForm(Owner)
{
    SimulationRunning = SimulationStopped = 0;
    FeedMetabEnergy = Vector (FEED_NUTRIENTS);
}
//---------------------------------------------------------------------------
// Validates user input for variables with lower and upper limits:
void TMainWindow::ValidateValue (TEdit *Edit, double min, double max){
    double Value = atof (Edit->Text.c_str());
    if (Value < min)
        Edit->Text = min;
    if (Value > max)
        Edit->Text = max;
}
//---------------------------------------------------------------------------
// Main window events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FormCreate(TObject *Sender)
{
    try {
        HiddenImage->Picture->LoadFromFile ("PIG.BMP");
    } catch (...) {}
    HiddenImage->Cursor = crFlyCursor;
    DietGrid->DefaultColWidth = (DietGrid->Width + 10) / 8;
    DietGrid->ColWidths[0] = DietGrid->Width - 6 * DietGrid->DefaultColWidth - 11;
    DietGrid->DefaultRowHeight = (DietGrid->Height) / 8 - 1;
    // Initialize variables that don't depend on data:
    InitEnvTab();
    InitFeedTab();
    InitHeatTab();
    InitMetaTab();
    // Read the default data from DEFAULT.SWI or set the defaults:
    DataTransfer.SetFileNames ("*");
    DataTransfer.OpenSwineSimFile();
    // Initialize the variables that depend on data:
    UpdateEnvGraph ();
    UpdatePhaseFields ();
    UpdatePhaseGraph ();
    UpdateGrowthGraph ();
    // Other:
    HiddenPanel->Width = (ClientWidth > HiddenPanel->Left + 40 ? ClientWidth - HiddenPanel->Left - 8 :
0);
    Application->OnIdle = IdleEvent;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FormResize(TObject *Sender)
{
    EnvGraph->Visible = false;
    EnvGraph->Height = EnvTab->Height - EnvGraph->Top;
    EnvGraph->Width = EnvTab->Width;
    EnvGraph->Visible = true;
    DietGrid->Width = FeedTab->Width;
```

```
    PhaseGraph->Visible = false;
    PhaseGraph->Height = FeedTab->Height - PhaseGraph->Top;
    PhaseGraph->Width = FeedTab->Width;
    PhaseGraph->Visible = true;
    SimGraph->Visible = false;
    SimGraph->Height = SimTab->Height - SimGraph->Top;
    SimGraph->Width = SimTab->Width;
    SimGraph->Visible = true;
    GrowthGraph->Visible = false;
    GrowthGraph->Height = GrowthTab->Height - GrowthGraph->Top;
    GrowthGraph->Width = GrowthTab->Width;
    GrowthGraph->Visible = true;
    HiddenPanel->Width = (ClientWidth > HiddenPanel->Left + 40 ? ClientWidth - HiddenPanel->Left - 8 :
0);
    CirandaMemo->Width = (ExpertTab->Width > CirandaMemo->Left + 80 ? ExpertTab->Width - CirandaMemo-
>Left - 8 : 0);
    CirandaMemo->Height = (ExpertTab->Height > CirandaMemo->Top + 80 ? ExpertTab->Height - CirandaMemo-
>Top - 8 : 0);
    CirandaMemo->Font->Name = "Symbol";
]
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PageControlChange(TObject *Sender)
{
    EnvGraph->Visible = false;
    EnvGraph->Height = EnvTab->Height - EnvGraph->Top;
    EnvGraph->Width = EnvTab->Width;
    EnvGraph->Visible = true;
    PhaseGraph->Visible = false;
    DietGrid->Width = FeedTab->Width;
    PhaseGraph->Height = FeedTab->Height - PhaseGraph->Top;
    PhaseGraph->Width = FeedTab->Width;
    PhaseGraph->Visible = true;
    SimGraph->Visible = false;
    SimGraph->Height = SimTab->Height - SimGraph->Top;
    SimGraph->Width = SimTab->Width;
    SimGraph->Visible = true;
    GrowthGraph->Visible = false;
    GrowthGraph->Height = GrowthTab->Height - GrowthGraph->Top;
    GrowthGraph->Width = GrowthTab->Width;
    GrowthGraph->Visible = true;
    HiddenPanel->Width = (ClientWidth > HiddenPanel->Left + 40 ? ClientWidth - HiddenPanel->Left - 8 :
0);
    CirandaMemo->Width = (ExpertTab->Width > CirandaMemo->Left + 80 ? ExpertTab->Width - CirandaMemo-
>Left - 8 : 0);
    CirandaMemo->Height = (ExpertTab->Height > CirandaMemo->Top + 80 ? ExpertTab->Height - CirandaMemo-
>Top - 8 : 0);
    CirandaMemo->Font->Name = "Symbol";
    HiddenFlavio->Visible = false;
    Matrix D (POOL_NUTRIENTS, FEED_NUTRIENTS);
    Vector E (POOL_NUTRIENTS);
    for (int i = 0; i < POOL_NUTRIENTS; i ++) {
        for (int j = 0; j < FEED_NUTRIENTS; j ++)
            D[i][j] = atof (DigestionGrid->Cells[j+1][i+1].c_str());
    }
    E [POOL_PROTEIN] = atof (EMP->Text.c_str());
    E [POOL_FAT] = atof (EML->Text.c_str());
    E [POOL_CARBS] = atof (EMG->Text.c_str());
    FeedMetabEnergy = E * D;
}
//---------------------------------------------------------------------------
// Main menu options
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FileNewClick(TObject *Sender)
{
    FileOpenDialog->FileName = "";
    FileSaveDialog->FileName = "";
    DataTransfer.SetFileNames ("");
    DataTransfer.OpenSwineSimFile ();
]
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FileOpenClick(TObject *Sender)
{
    if (! FileOpenDialog->Execute ())
        return;
    FileSaveDialog->FileName = FileOpenDialog->FileName;
    DataTransfer.SetFileNames (FileOpenDialog->FileName);
    DataTransfer.OpenSwineSimFile ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FileSaveClick(TObject *Sender)
{
    if (FileSaveDialog->FileName=="*"||access(FileSaveDialog->FileName.c_str(),6))
        if (! FileSaveDialog->Execute ())
            return;
    FileOpenDialog->FileName = FileSaveDialog->FileName;
    DataTransfer.SetFileNames (FileSaveDialog->FileName);
    DataTransfer.SaveSwineSimFile ();
```

```
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FileSaveAsClick(TObject *Sender)
{
    if (! FileSaveDialog->Execute ())
        return;
    FileOpenDialog->FileName = FileSaveDialog->FileName;
    DataTransfer.SetFileNames (FileSaveDialog->FileName);
    DataTransfer.SaveSwineSimFile ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FilePrintClick(TObject *Sender)
{
    if (PrintDialog->Execute()) {
      }
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FilePrinterSetupClick(TObject *Sender)
{
    PrinterSetupDialog->Execute();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FileExitClick(TObject *Sender)
{
    Close ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HelpAboutClick(TObject *Sender)
{
    AboutBox->ShowModal();
}
//---------------------------------------------------------------------------
// Simulation tab events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimStartAgeExit(TObject *Sender)
{
    double StartAge = atof (SimStartAge->Text.c_str());
    if (StartAge < 0)
        SimStartAge->Text = 0;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimStartWeightExit(TObject *Sender)
{
    double StartWeight = atof (SimStartWeight->Text.c_str());
    if (StartWeight < 0)
        SimStartWeight->Text = 0;
    else if (atof (SimFinishWeight->Text.c_str()) < StartWeight)
        SimFinishWeight->Text = StartWeight;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::MetFatLeanRatioExit(TObject *Sender)
{
    if (atof (MetFatLeanRatio->Text.c_str()) < atof (kFL->Text.c_str()))
        MetFatLeanRatio->Text = atof (kFL->Text.c_str());
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimRbFinishDaysClick(TObject *Sender)
{
    SimSimulatedDays->Enabled = true;
    SimFinishWeight->Enabled = false;
    SimDaysLabel->Enabled = true;
    SimKgLabel->Enabled = false;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimSimulatedDaysExit(TObject *Sender)
{
    double SimulatedDays = atof (SimSimulatedDays->Text.c_str());
    if (SimulatedDays < 0)
        SimSimulatedDays->Text = SimulatedDays = 0;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimRbFinishWeightClick(TObject *Sender)
{
    SimFinishWeight->Enabled = true;
    SimSimulatedDays->Enabled = false;
    SimKgLabel->Enabled = true;
    SimDaysLabel->Enabled = false;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimFinishWeightExit(TObject *Sender)
{
    double FinishWeight = atof (SimFinishWeight->Text.c_str());
    if (FinishWeight < 0)
        SimFinishWeight->Text = FinishWeight = 0;
    if (atof (SimStartWeight->Text.c_str()) > FinishWeight)
        SimStartWeight->Text = FinishWeight;
}
```

```
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimTimeStepExit(TObject *Sender)
{
    if (atof (SimTimeStep->Text.c_str()) <= 0)
        SimTimeStep->Text = 24.0 / STEPS_PER_DAY;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimBtnRefreshClick(TObject *Sender)
{
    UpdateSimGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimBtnPrintClick(TObject *Sender)
{
    UpdateSimGraph();
    SimGraph->Visible = false;
    SimGraph->Height = SimGraph->Width = 2000;
    SimGraph->DrawMode = gphPrint;
    SimGraph->Height = SimTab->Height - SimGraph->Top;
    SimGraph->Width = SimTab->Width;
    SimGraph->Visible = true;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimBtnAcceptClick(TObject *Sender)
{
    DataTransfer.ReadSimulationFromScreen ();
}
//---------------------------------------------------------------------------

void __fastcall TMainWindow::SimBtnCancelClick(TObject *Sender)
{
    DataTransfer.WriteSimulationToScreen ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimBtnStartClick(TObject *Sender)
{
    if (SimulationRunning) {
        if (SimulationStopped) { // simulation paused, pressed stop
            SimBtnStart->Caption = "Continue";
            SimBtnReset->Caption = "Reset";
            SimulationRunning = 0;
            SimulationStopped = 1;
            StatusBar->SimpleText = "Simulation stopped";
        } else { // simulation running, pressed stop
            DataTransfer.ExecuteExpertSystem ();
            UpdateSimGraph ();
            SimBtnStart->Caption = "Continue";
            SimBtnReset->Caption = "Reset";
            SimulationRunning = 0;
            SimulationStopped = 1;
            StatusBar->SimpleText = "Simulation stopped";
        }
    } else {
        if (SimulationStopped) { // simulation stopped, pressed continue
            if (Simulation.ContinueSimulation ()) {
                SimBtnStart->Caption = "Stop";
                SimBtnReset->Caption = "Pause";
                SimulationRunning = 1;
                SimulationStopped = 0;
                StatusBar->SimpleText = "Simulation running";
            }
        } else { // simulation starting, pressed start
            if (Simulation.StartSimulation ()) {
                SimBtnStart->Caption = "Stop";
                SimBtnReset->Caption = "Pause";
                SimulationRunning = 1;
                SimulationStopped = 0;
                StatusBar->SimpleText = "Simulation running";
            }
        }
    }
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimBtnResetClick(TObject *Sender)
{
    if (SimulationRunning) {
        if (SimulationStopped) {// simulation paused, pressed resume
            SimBtnStart->Caption = "Stop";
            SimBtnReset->Caption = "Pause";
            SimulationRunning = 1;
            SimulationStopped = 0;
            StatusBar->SimpleText = "Simulation running";
        } else {// simulation running, pressed pause
            DataTransfer.ExecuteExpertSystem ();
            UpdateSimGraph ();
            SimBtnStart->Caption = "Stop";
            SimBtnReset->Caption = "Resume";
```

```
            SimulationRunning = 1;
            SimulationStopped = 1;
            StatusBar->SimpleText = "Simulation paused";
        }
    } else { // simulation stopped or starting, pressed reset
        DataTransfer.SaveSimulationFile ();
        SimBtnStart->Caption = "Start";
        SimBtnReset->Caption = "Reset";
        SimulationRunning = 0;
        SimulationStopped = 0;
        StatusBar->SimpleText = "Simulation reset";
    }
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::SimGraphTypeClick(TObject *Sender)
{
    UpdateSimGraph ();
}
//---------------------------------------------------------------------------
// Expert system tab events
//---------------------------------------------------------------------------
// Environment tab events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::Ta_minExit(TObject *Sender)
{
    ValidateValue (Ta_min, -50, 50);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tTa_minExit(TObject *Sender)
{
    ValidateEnvTime (tTa_min, 0, atof (tTa_max->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::Ta_maxExit(TObject *Sender)
{
    ValidateValue (Ta_max, -50, 50);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tTa_maxExit(TObject *Sender)
{
    ValidateEnvTime (tTa_max, atof (tTa_min->Text.c_str()), 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::Tr_minExit(TObject *Sender)
{
    ValidateValue (Tr_min, -50, 50);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tTr_minExit(TObject *Sender)
{
    ValidateEnvTime (tTr_min, 0, atof (tTr_max->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::Tr_maxExit(TObject *Sender)
{
    ValidateValue (Tr_max, -50, 50);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tTr_maxExit(TObject *Sender)
{
    ValidateEnvTime (tTr_max, atof (tTr_min->Text.c_str()), 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::Tf_minExit(TObject *Sender)
{
    ValidateValue (Tf_min, -50, 50);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tTf_minExit(TObject *Sender)
{
    ValidateEnvTime (tTf_min, 0, atof (tTf_max->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::Tf_maxExit(TObject *Sender)
{
    ValidateValue (Tf_max, -50, 50);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tTf_maxExit(TObject *Sender)
{
    ValidateEnvTime (tTf_max, atof (tTf_min->Text.c_str()), 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::phia_maxExit(TObject *Sender)
{
    ValidateValue (phia_max, 0, 100);
}
```

```
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tphia_maxExit(TObject *Sender)
{
   ValidateEnvTime (tphia_max, 0, atof (tphia_min->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::phia_minExit(TObject *Sender)
{
   ValidateValue (phia_min, 0, 100);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tphia_minExit(TObject *Sender)
{
   ValidateEnvTime (tphia_min, atof (tphia_max->Text.c_str()), 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::va_min1Exit(TObject *Sender)
{
   ValidateValue (va_min1, 0, 20);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tVa_min1Exit(TObject *Sender)
{
   ValidateEnvTime (tVa_min1, 0, atof (tVa_max1->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::va_max1Exit(TObject *Sender)
{
   ValidateValue (va_max1, 0, 20);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tVa_max1Exit(TObject *Sender)
{
   ValidateEnvTime (tVa_max1,
      atof (tVa_min1->Text.c_str()), atof (tVa_min2->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::va_min2Exit(TObject *Sender)
{
   ValidateValue (va_min2, 0, 20);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tVa_min2Exit(TObject *Sender)
{
   ValidateEnvTime (tVa_min2,
      atof (tVa_max1->Text.c_str()), atof (tVa_max2->Text.c_str()));
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::va_max2Exit(TObject *Sender)
{
   ValidateValue (va_max2, 0, 20);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tVa_max2Exit(TObject *Sender)
{
   ValidateEnvTime (tVa_max2, atof (tVa_min2->Text.c_str()), 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::vafExit(TObject *Sender)
{
   ValidateValue (vaf, 0, 20);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tVa_onExit(TObject *Sender)
{
   ValidateEnvTime (tVa_on, 0, 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::tVa_offExit(TObject *Sender)
{
   ValidateEnvTime (tVa_off, 0, 24);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::EnvBtnRefreshClick(TObject *Sender)
{
   UpdateEnvGraph();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::EnvBtnPrintClick(TObject *Sender)
{
   UpdateEnvGraph();
   EnvGraph->Visible = false;
   EnvGraph->Height = EnvGraph->Width = 2000;
   EnvGraph->DrawMode = gphPrint;
   EnvGraph->Height = EnvTab->Height - EnvGraph->Top;
   EnvGraph->Width = EnvTab->Width;
   EnvGraph->Visible = true;
```

```
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::EnvBtnAcceptClick(TObject *Sender)
{
    DataTransfer.ReadEnvironmentFromScreen ();
    UpdateEnvGraph();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::EnvBtnCancelClick(TObject *Sender)
{
    DataTransfer.WriteEnvironmentToScreen ();
    UpdateEnvGraph();
}
//---------------------------------------------------------------------------
// Feeding program tab events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FeedUseEfficiency_Exit(TObject *Sender)
{
    double UseEfficiency = atof (FeedUseEfficiency->Text.c_str());
    if ((UseEfficiency <= 0) || (UseEfficiency > 1))
        FeedUseEfficiency->Text = "1";
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietName_Exit(TObject *Sender)
{
    DietName->Text = DietName->Text.Trim();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietBtnAddClick(TObject *Sender)
{
    DietAdd ();
    DietGrid->Repaint();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietBtnInsertClick(TObject *Sender)
{
    DietInsert ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietBtnDeleteClick(TObject *Sender)
{
    DietDelete ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietBtnRenameClick(TObject *Sender)
{
    DietRename ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseTransitionTypeClick(TObject *Sender)
{
    UpdatePhaseGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FeedBtnRefreshClick(TObject *Sender)
{
    UpdatePhaseGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FeedBtnPrintClick(TObject *Sender)
{
    UpdatePhaseGraph ();
    PhaseGraph->DrawMode = gphNoAction;
    PhaseGraph->Palette = gphGrayscale;
    PhaseGraph->ThisSet = 1; PhaseGraph->ColorData = 11;
    PhaseGraph->ThisSet = 2; PhaseGraph->ColorData = 7;
    PhaseGraph->ThisSet = 3; PhaseGraph->ColorData = 4;
    PhaseGraph->ThisSet = 4; PhaseGraph->ColorData = 1;
    PhaseGraph->ThisSet = 5; PhaseGraph->ColorData = 13;
    PhaseGraph->ThisSet = 6; PhaseGraph->ColorData = 9;
    PhaseGraph->ThisSet = 7; PhaseGraph->ColorData = 6;
    PhaseGraph->ThisSet = 8; PhaseGraph->ColorData = 3;
    PhaseGraph->ThisSet = 9; PhaseGraph->ColorData = 10;
    PhaseGraph->ThisSet = 10; PhaseGraph->ColorData = 2;
    PhaseGraph->Visible = false;
    PhaseGraph->Height = PhaseGraph->Width = 1200;
    PhaseGraph->DrawMode = gphPrint;
    PhaseGraph->Height = FeedTab->Height - PhaseGraph->Top;
    PhaseGraph->Width = FeedTab->Width;
    PhaseGraph->Visible = true;
    PhaseGraph->Palette = gphDefault;
    PhaseGraph->ThisSet = 1; PhaseGraph->ColorData = 14;
    PhaseGraph->ThisSet = 2; PhaseGraph->ColorData = 10;
    PhaseGraph->ThisSet = 3; PhaseGraph->ColorData = 12;
    PhaseGraph->ThisSet = 4; PhaseGraph->ColorData = 9;
    PhaseGraph->ThisSet = 5; PhaseGraph->ColorData = 6;
    PhaseGraph->ThisSet = 6; PhaseGraph->ColorData = 3;
```

```
    PhaseGraph->ThisSet = 7; PhaseGraph->ColorData = 5;
    PhaseGraph->ThisSet = 8; PhaseGraph->ColorData = 2;
    PhaseGraph->ThisSet = 9; PhaseGraph->ColorData = 4;
    PhaseGraph->ThisSet = 10; PhaseGraph->ColorData = 1;
    PhaseGraph->DrawMode = gphBlit;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FeedBtnAcceptClick(TObject *Sender)
{
    DataTransfer.ReadFeedingProgramFromScreen ();
    UpdatePhaseGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::FeedBtnCancelClick(TObject *Sender)
{
    DataTransfer.WriteFeedingProgramToScreen ();
    UpdatePhaseGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseScrollScroll(TObject *Sender,
    TScrollCode ScrollCode, int &ScrollPos)
{
    PhaseCurrent->Caption = ScrollPos;
    UpdatePhaseFields ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseNameExit(TObject *Sender)
{
    strncpy (Phase[atoi(PhaseCurrent->Caption.c_str())-1].Name,
        PhaseName->Text.c_str(), NAME_SIZE-1);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseDietChange(TObject *Sender)
{
    Phase[atoi(PhaseCurrent->Caption.c_str())-1].DietNumber=PhaseDiet->ItemIndex;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseEnabledClick(TObject *Sender)
{
    Phase[atoi(PhaseCurrent->Caption.c_str())-1].Enabled = PhaseEnabled->Checked;
    UpdatePhaseFields ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseAdLibClick(TObject *Sender)
{
    Phase[atoi(PhaseCurrent->Caption.c_str())-1].AdLib = PhaseAdLib->Checked;
    UpdatePhaseFields ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseStartAgeExit(TObject *Sender)
{
    ValidatePhaseAge (atoi(PhaseCurrent->Caption.c_str())-1);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseStartWeightExit(TObject *Sender)
{
    ValidatePhaseWeight (atoi(PhaseCurrent->Caption.c_str())-1);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseInitialFeedExit(TObject *Sender)
{
    ValidatePhaseInitialFeed (atoi(PhaseCurrent->Caption.c_str())-1);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseWeeklyIncExit(TObject *Sender)
{
    Phase[atoi(PhaseCurrent->Caption.c_str())-1].WeeklyInc =
        atof (PhaseWeeklyInc->Text.c_str());
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseUpDownActionClick(TObject *Sender,
    TUDBtnType Button)
{
    PhaseActionEnabled->SetFocus();
    PhaseAction->Caption = PhaseUpDownAction->Position;
    UpdatePhaseFields ();

}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseActionEnabledClick(TObject *Sender)
{
    Phase[atoi(PhaseCurrent->Caption.c_str())-1].Action[atoi(PhaseAction->
        Caption.c_str())-1].Enabled = PhaseActionEnabled->Checked;
    UpdatePhaseFields ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseTimeOfActionExit(TObject *Sender)
```

```
{
    ValidatePhaseTimeOfAction (atoi(PhaseCurrent->Caption.c_str())-1,
        atoi(PhaseAction->Caption.c_str())-1);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseActionTypeChange(TObject *Sender)
{
    Phase[atoi(PhaseCurrent->Caption.c_str())-1].Action[atoi(PhaseAction->
        Caption.c_str())-1].Type = (TActionType) PhaseActionType->ItemIndex;
    UpdatePhaseFields ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::PhaseActionAmountExit(TObject *Sender)
{
    ValidatePhaseActionAmount (atoi(PhaseCurrent->Caption.c_str())-1,
        atoi(PhaseAction->Caption.c_str())-1);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietGrid_SelectCell(TObject *Sender, long Col,
    long Row, bool &CanSelect)
{
    DietName->Text = DietGrid->Cells[Col][0];
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::DietGridSetEditText(TObject *Sender, long ACol, long ARow,
    const AnsiString Value)
{
    Vector F (FEED_NUTRIENTS);
    for (int j = 0; j < FEED_NUTRIENTS; j ++) {
        F[j] = atof (DietGrid->Cells[ACol][j+2].c_str());
    }
    MainWindow->DietGrid->Cells[ACol][1] =
        AnsiString::FloatToStrF (FeedMetabEnergy * F / 1000, AnsiString::sffFixed, 5, 3);
}
//---------------------------------------------------------------------------
// Heat balance tab events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::BuildRbFloorTypeClick(TObject *Sender)
{
    BuildFloorType->Enabled = true;
    Cf0->Enabled = false;
    Label82->Enabled = false;
    Label130->Enabled = false;
    Cf0->Text = HeatBalance.Cf0_List[BuildFloorType->ItemIndex];
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::BuildRbFloorConductanceClick(TObject *Sender)
{
    Cf0->Enabled = true;
    Label82->Enabled = true;
    Label130->Enabled = true;
    BuildFloorType->Enabled = false;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::BuildFloorTypeChange(TObject *Sender)
{
    Cf0->Text = HeatBalance.Cf0_List[BuildFloorType->ItemIndex];
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::EnvFeedTempEqualAirClick(TObject *Sender)
{
     Ti->Enabled = ! EnvFeedTempEqualAir->ItemIndex;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HeatBtnAcceptClick(TObject *Sender)
{
    DataTransfer.ReadHeatBalanceFromScreen ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HeatBtnCancelClick(TObject *Sender)
{
    DataTransfer.WriteHeatBalanceToScreen ();
}
//---------------------------------------------------------------------------
// Metabolism tab events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::MetBtnAcceptClick(TObject *Sender)
{
    DataTransfer.ReadMetabolismFromScreen ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::MetBtnCancelClick(TObject *Sender)
{
    DataTransfer.WriteMetabolismToScreen ();
}
//---------------------------------------------------------------------------
// Growth curves tab events
```

```
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthRbl1Click(TObject *Sender)
{
    GrowthGroup1->Enabled = true;
        Growth1_kL->Enabled = true;
        Growth1_kF->Enabled = true;
    GrowthGroup2->Enabled = false;
        Growth2_WmaxL->Enabled = false;
        Growth2_aL->Enabled = false;
        Growth2_tmaxL->Enabled = false;
        Growth2_WmaxF->Enabled = false;
        Growth2_aF->Enabled = false;
        Growth2_tmaxF->Enabled = false;
        Growth2_gestation->Enabled = false;
        Growth2_PhysAge->Enabled = false;
    GrowthGroup3->Enabled = false;
    UpdateGrowthGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthRb2Click(TObject *Sender)
{
    GrowthGroup2->Enabled = true;
        Growth2_WmaxL->Enabled = true;
        Growth2_aL->Enabled = true;
        Growth2_tmaxL->Enabled = true;
        Growth2_WmaxF->Enabled = true;
        Growth2_aF->Enabled = true;
        Growth2_tmaxF->Enabled = true;
        Growth2_gestation->Enabled = true;
        Growth2_PhysAge->Enabled = true;
    GrowthGroup1->Enabled = false;
        Growth1_kL->Enabled = false;
        Growth1_kF->Enabled = false;
    GrowthGroup3->Enabled = false;
    UpdateGrowthGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthRb3Click(TObject *Sender)
{
    GrowthGroup3->Enabled = true;
    GrowthGroup1->Enabled = false;
        Growth1_kL->Enabled = false;
        Growth1_kF->Enabled = false;
    GrowthGroup2->Enabled = false;
        Growth2_WmaxL->Enabled = false;
        Growth2_aL->Enabled = false;
        Growth2_tmaxL->Enabled = false;
        Growth2_WmaxF->Enabled = false;
        Growth2_aF->Enabled = false;
        Growth2_tmaxF->Enabled = false;
        Growth2_gestation->Enabled = false;
        Growth2_PhysAge->Enabled = false;
    UpdateGrowthGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthBtnRefreshClick(TObject *Sender)
{
    UpdateGrowthGraph ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthBtnPrintClick(TObject *Sender)
{
    UpdateGrowthGraph ();
    GrowthGraph->Visible = false;
    GrowthGraph->Height = GrowthGraph->Width = 2000;
    GrowthGraph->DrawMode = gphPrint;
    GrowthGraph->Height = GrowthTab->Height - GrowthGraph->Top;
    GrowthGraph->Width = GrowthTab->Width;
    GrowthGraph->Visible = true;
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthBtnAcceptClick(TObject *Sender)
{
    DataTransfer.ReadMetabolismFromScreen ();
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::GrowthBtnCancelClick(TObject *Sender)
{
    DataTransfer.WriteMetabolismToScreen ();
}
//---------------------------------------------------------------------------
// Easter egg events
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HiddenImageClick(TObject *Sender)
{
    if (AboutBox->PigSound->Mode == mpPlaying)
        AboutBox->PigSound->Pause ();
```

```
    else {
        try {
            AboutBox->PigSound->Play ();
        } catch (EMCIDeviceError &E) {}
    }
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::CirandaMemoDblClick(TObject *Sender)
{
    static int count = 0;
    switch (count) {
        case 0:
        case 2:
            if (AboutBox->PigSound->Mode == mpPlaying)
                count ++;
            else
                count = 0;
            break;
        case 1:
        case 3:
            if (AboutBox->PigSound->Mode != mpPlaying)
                count ++;
            else
                count = 0;
            break;
        case 4:
            if (AboutBox->PigSound->Mode == mpPlaying) {
                CirandaMemo->Font->Name = "Times New Roman";
                count ++;
                if (! HeatCircleShape->Visible)
                    HeatCircleShape->Visible = true;
                else if (! HeatRedShape->Visible)
                    HeatRedShape->Visible = true;
                else if (! HeatGreenShape->Visible)
                    HeatGreenShape->Visible = true;
                else if (! HeatBlueShape->Visible)
                    HeatBlueShape->Visible = true;
                else {
                    HeatCircleShape->Visible = false;
                    HeatRedShape->Visible = false;
                    HeatGreenShape->Visible = false;
                    HeatBlueShape->Visible = false;
                }
            } else
                count = 0;
            break;
        default:
            CirandaMemo->Font->Name = "Symbol";
            count = 0;
    }
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HeatRedShapeDragOver(TObject *Sender,
    TObject *Source, int X, int Y, TDragState State, bool &Accept)
{
    Accept = true;
    HeatRedShape->Brush->Color = (Graphics::TColor) (0x1F + 0x20 *
        ((8 * X) / HeatRedShape->Width));
    HeatCircleShape->Brush->Color = (Graphics::TColor) ((0xFFFF00 &
        HeatCircleShape->Brush->Color) + HeatRedShape->Brush->Color);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HeatGreenShapeDragOver(TObject *Sender,
    TObject *Source, int X, int Y, TDragState State, bool &Accept)
{
    Accept = true;
    HeatGreenShape->Brush->Color = (Graphics::TColor) (0x100 * (0x1F + 0x20 *
        ((8 * X) / HeatGreenShape->Width)));
    HeatCircleShape->Brush->Color = (Graphics::TColor) ((0xFF00FF &
        HeatCircleShape->Brush->Color) + HeatGreenShape->Brush->Color);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HeatBlueShapeDragOver(TObject *Sender,
    TObject *Source, int X, int Y, TDragState State, bool &Accept)
{
    Accept = true;
    HeatBlueShape->Brush->Color = (Graphics::TColor) (0x10000 * (0x1F + 0x20 *
        ((8 * X) / HeatBlueShape->Width)));
    HeatCircleShape->Brush->Color = (Graphics::TColor) ((0x00FFFF &
        HeatCircleShape->Brush->Color) + HeatBlueShape->Brush->Color);
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HeatCircleShapeMouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if (HeatCircleShape->Brush->Color == 0xFFBF1F) {
        HeatCircleShape->Visible = false;
```

```
        HeatRedShape->Visible = false;
        HeatGreenShape->Visible = false;
        HeatBlueShape->Visible = false;
        HiddenFlavio->Visible = true;
    }
}
//---------------------------------------------------------------------------
void __fastcall TMainWindow::HiddenFlavioClick(TObject *Sender)
{
        HiddenFlavio->Visible = false;
}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
```

SwineSimMain.dfm

Binary file containing the layout of the Main Window form.

SwineSimDataTransfer.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimDataTransferH
#define SwineSimDataTransferH

#include <vcl\inifiles.hpp>
//---------------------------------------------------------------------------

// This structure is used for reading and writing variables to a file:
struct TVariableDescription {
    double * Address; // address of the variable in memory
    TEdit * Edit; // Edit control for the variable on the GUI
    char * Name; // name of the variable in the file
    double Default; // default value when the variable is not found in the file
};

// This class takes care of data transfer between the variables in memory,
// the screen values shown in the GUI, and the variables stored in a disk file.
// Transfers are always between memory and screen or between memory and disk:
class TDataTransfer {
public:
    void SetFileNames (System::AnsiString FileName);
    void OpenSwineSimFile (void);
    void SaveSwineSimFile (void);
    void SetEnvironmentDescription (void);
    void SetHeatBalanceDescription (void);
    void SetMetabolismDescription (void);
    void ReadSimulationFromScreen (void);
    void ReadEnvironmentFromScreen (void);
    void ReadFeedingProgramFromScreen (void);
    void ReadHeatBalanceFromScreen (void);
    void ReadMetabolismFromScreen (void);
    void WriteSimulationToScreen (void);
    void WriteEnvironmentToScreen (void);
    void WriteFeedingProgramToScreen (void);
    void WriteHeatBalanceToScreen (void);
    void WriteMetabolismToScreen (void);
    void SaveSimulationFile (void);
    void ExecuteExpertSystem (void);
private:
    System::AnsiString InputFileName;
    System::AnsiString OutputFileName;
    System::AnsiString DailyFileName;
    static TVariableDescription EnvironmentDescription [38];
    static TVariableDescription HeatBalanceDescription [29];
    static TVariableDescription MetabolismDescription [52];
    static const double DigestionMatrixDefaults [POOL_NUTRIENTS+2][FEED_NUTRIENTS];
    void ReadSimulationFromFile (TIniFile * IniFile);
    void ReadEnvironmentFromFile (TIniFile * IniFile);
    void ReadFeedingProgramFromFile (TIniFile * IniFile);
    void ReadHeatBalanceFromFile (TIniFile * IniFile);
    void ReadMetabolismFromFile (TIniFile * IniFile);
    void WriteSimulationToFile (TIniFile * IniFile);
    void WriteEnvironmentToFile (TIniFile * IniFile);
    void WriteFeedingProgramToFile (TIniFile * IniFile);
```

```
    void WriteHeatBalanceToFile (TIniFile * IniFile);
    void WriteMetabolismToFile (TIniFile * IniFile);
    void WriteOutputTitle (FILE * f);
    void WriteOutputLine (FILE * f, TSimulatedData & SimData);
};

// This variable implements the TDataTransfer class:
extern TDataTransfer DataTransfer;
//---------------------------------------------------------------------------
#endif
```

SwineSimDataTransfer.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include <dir.h>
#include "SwineSim.h"
#include "SwineSimDataTransfer.h"
//---------------------------------------------------------------------------

// This variable implements the TDataTransfer class:
TDataTransfer DataTransfer;

void TDataTransfer::SetFileNames (System::AnsiString FileName) {
    char path[MAXPATH];
    char drive[MAXDRIVE];
    char dir[MAXDIR];
    char name[MAXFILE];
    if (FileName == "") { // no file name given
        fnsplit (Application->ExeName.c_str(), drive, dir, NULL, NULL);
        fnmerge (path, drive, dir, "\\Datafiles\\Default", "*.swi");
        InputFileName = path;
        fnmerge (path, drive, dir, "\\Datafiles\\Default", "*.swr");
        OutputFileName = path;
        fnmerge (path, drive, dir, "\\Datafiles\\Default", "*.swd");
        DailyFileName = path;
    } else {
        InputFileName = FileName;
        fnsplit (FileName.c_str(), drive, dir, name, NULL);
        fnmerge (path, drive, dir, name, "*.swr");
        OutputFileName = path;
        fnmerge (path, drive, dir, name, "*.swd");
        DailyFileName = path;
    }
}

// Read all variables from a disk file to memory to the GUI:
void TDataTransfer::OpenSwineSimFile (void) {
    TIniFile * IniFile;
    IniFile = new TIniFile (InputFileName);
    memset (DailyTotals, 0, sizeof (DailyTotals));
    memset (DailySimData, 0, sizeof (DailySimData));
    memset (SimulatedData, 0, sizeof (SimulatedData));
    ReadSimulationFromFile (IniFile);
    WriteSimulationToScreen ();
    ReadEnvironmentFromFile (IniFile);
    WriteEnvironmentToScreen ();
    ReadHeatBalanceFromFile (IniFile);
    WriteHeatBalanceToScreen ();
    ReadMetabolismFromFile (IniFile);
    WriteMetabolismToScreen ();
    ReadFeedingProgramFromFile (IniFile);
    WriteFeedingProgramToScreen ();
    Ex.StartExpert (MainWindow->ExpertMemo->Lines);
    Ex.FinishExpert();
    MainWindow->SimBtnStart->Caption = "Start";
    MainWindow->SimBtnReset->Caption = "Reset";
    MainWindow->SimulationRunning = 0;
    MainWindow->SimulationStopped = 0;
    MainWindow->StatusBar->SimpleText = "File loaded";
    delete (IniFile);
}

// Write all variables from the GUI to memory to a disk file:
void TDataTransfer::SaveSwineSimFile (void) {
    TIniFile * IniFile;
    IniFile = new TIniFile (InputFileName);
    ReadSimulationFromScreen ();
    WriteSimulationToFile (IniFile);
    ReadEnvironmentFromScreen ();
```

```
    WriteEnvironmentToFile (IniFile);
    ReadHeatBalanceFromScreen ();
    WriteHeatBalanceToFile (IniFile);
    ReadMetabolismFromScreen ();
    WriteMetabolismToFile (IniFile);
    ReadFeedingProgramFromScreen ();
    WriteFeedingProgramToFile (IniFile);
    MainWindow->StatusBar->SimpleText = "File saved";
    delete (IniFile);
}

void TDataTransfer::WriteOutputTitle (FILE * f) {
    int j;
    fprintf (f, "Time\tAge\tTa\tTr\tTf\tphia\tva\tI\tTb\tTss\tTse\tTsf\tTh\tTe\tTc\tqP\tqL\tqss\t"
        "qse\tqsf\tqv\tqi\tqsec\tqser\tqsee\tqssc\tqssr\tqsse\tqsss\tqpm\tqpq\tqpd\tqpl\tqpf\t");
    for (j = 0; j < POOL_NUTRIENTS; j ++)
        fprintf (f, "Np[%d]\t", j);
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        fprintf (f, "Ni[%d]\t", j);
    fprintf (f, "dWlt\tdWft\tWb\tWe\tWlt\tWft\tDdWb\tDGF\tDRs\tDRc\tDRw\tRf\tRa\tRl\tPhase\tdWb\t"
        "FL\n");
}

void TDataTransfer::WriteOutputLine (FILE * f, TSimulatedData & SimData) {
    int j;
    fprintf (f, "%f\t", SimData.Time);
    fprintf (f, "%f\t", SimData.Age);
    fprintf (f, "%f\t", SimData.Ta);
    fprintf (f, "%f\t", SimData.Tr);
    fprintf (f, "%f\t", SimData.Tf);
    fprintf (f, "%f\t", SimData.phia);
    fprintf (f, "%f\t", SimData.va);
    fprintf (f, "%f\t", SimData.I);
    fprintf (f, "%f\t", SimData.Tb);
    fprintf (f, "%f\t", SimData.Tss);
    fprintf (f, "%f\t", SimData.Tse);
    fprintf (f, "%f\t", SimData.Tsf);
    fprintf (f, "%f\t", SimData.Th);
    fprintf (f, "%f\t", SimData.Te);
    fprintf (f, "%f\t", SimData.Tc);
    fprintf (f, "%f\t", SimData.qP);
    fprintf (f, "%f\t", SimData.qL);
    fprintf (f, "%f\t", SimData.qss);
    fprintf (f, "%f\t", SimData.qse);
    fprintf (f, "%f\t", SimData.qsf);
    fprintf (f, "%f\t", SimData.qv);
    fprintf (f, "%f\t", SimData.qi);
    fprintf (f, "%f\t", SimData.qsec);
    fprintf (f, "%f\t", SimData.qser);
    fprintf (f, "%f\t", SimData.qsee);
    fprintf (f, "%f\t", SimData.qssc);
    fprintf (f, "%f\t", SimData.qssr);
    fprintf (f, "%f\t", SimData.qsse);
    fprintf (f, "%f\t", SimData.qsss);
    fprintf (f, "%f\t", SimData.qpm);
    fprintf (f, "%f\t", SimData.qpq);
    fprintf (f, "%f\t", SimData.qpd);
    fprintf (f, "%f\t", SimData.qpl);
    fprintf (f, "%f\t", SimData.qpf);
    for (j = 0; j < POOL_NUTRIENTS; j ++) {
        fprintf (f, "%f\t", SimData.Np[j]);
    }
    for (j = 0; j < FEED_NUTRIENTS; j ++) {
        fprintf (f, "%f\t", SimData.Np[j]);
    }
    fprintf (f, "%f\t", SimData.dWlt);
    fprintf (f, "%f\t", SimData.dWft);
    fprintf (f, "%f\t", SimData.Wb);
    fprintf (f, "%f\t", SimData.We);
    fprintf (f, "%f\t", SimData.Wlt);
    fprintf (f, "%f\t", SimData.Wft);
    fprintf (f, "%f\t", SimData.DdWb);
    fprintf (f, "%f\t", SimData.DGF);
    fprintf (f, "%f\t", SimData.DRs);
    fprintf (f, "%f\t", SimData.DRc);
    fprintf (f, "%f\t", SimData.DRw);
    fprintf (f, "%f\t", SimData.Rf);
    fprintf (f, "%f\t", SimData.Ra);
    fprintf (f, "%f\t", SimData.Rl);
    fprintf (f, "%d\t", SimData.Phase);
    fprintf (f, "%f\t", SimData.dWb);
    fprintf (f, "%f\n", SimData.FL);
}

void TDataTransfer::SaveSimulationFile (void) {
    int i;
    FILE * f;
```

```
    if ((f = fopen (OutputFileName.c_str(), "wt")) != NULL) {
        fprintf (f, "Obs\t");
        WriteOutputTitle (f);
        for (i = 0; i <= STEPS_PER_DAY * 10; i ++) {
            if (SimulatedData[i].Wb == 0)
                break;
            fprintf (f, "%d\t", i);
            WriteOutputLine (f, SimulatedData[i]);
        }
        fclose (f);
    }
    if ((f = fopen (DailyFileName.c_str(), "wt")) != NULL) {
        fprintf (f, "Obs\t");
        WriteOutputTitle (f);
        for (i = 0; i <= MAX_RECORDED_DAYS; i ++) {
            if (DailySimData[i].Wb == 0)
                break;
            fprintf (f, "%d\t", i);
            WriteOutputLine (f, DailySimData[i]);
        }
        fclose (f);
    }
}

void TDataTransfer::ExecuteExpertSystem (void) {
    System::AnsiString ExpertLine;
    Ex.StartExpert (MainWindow->ExpertMemo->Lines);
    ExpertLine = "AdditionalHeat=";
    ExpertLine += (Metabolism.QPT > 0 ? Metabolism.QPT_heat / Metabolism.QPT : 0);
    Ex.ExecuteLine (ExpertLine);
    ExpertLine = "LeanGrowthRate=";
    ExpertLine += (Metabolism.dWltPT > 0 ? Metabolism.dWltT / Metabolism.dWltPT : 1);
    Ex.ExecuteLine (ExpertLine);
    ExpertLine = "FatGrowthRate=";
    ExpertLine += (Metabolism.dWftPT > 0 ? Metabolism.dWftT / Metabolism.dWftPT : 1);
    Ex.ExecuteLine (ExpertLine);
    ExpertLine = "GrowthRate=";
    ExpertLine += 24 * (Simulation.SimulationTime ?
        (Metabolism.Wb - Simulation.InitialWeight) / Simulation.SimulationTime : 0);
    Ex.ExecuteLine (ExpertLine);
    double x = HeatBalance.t_cold + HeatBalance.t_hot + HeatBalance.t_evap
        + HeatBalance.t_neutral;
    if (x > 0) {
        ExpertLine = "TimeCold=";
        ExpertLine += HeatBalance.t_cold / x;
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TimeNeutral=";
        ExpertLine += HeatBalance.t_neutral / x;
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TimeEvaporating=";
        ExpertLine += HeatBalance.t_evap / x;
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TimeHot=";
        ExpertLine += HeatBalance.t_hot / x;
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TempTimeCold=";
        ExpertLine += (HeatBalance.t_cold ?
            HeatBalance.TT_cold / HeatBalance.t_cold : HeatBalance.Tc);
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TempTimeNeutral=";
        ExpertLine += (HeatBalance.t_neutral ?
            HeatBalance.TT_neutral / HeatBalance.t_neutral : (HeatBalance.Tc + HeatBalance.Te) / 2);
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TempTimeEvaporating=";
        ExpertLine += (HeatBalance.t_evap ?
            HeatBalance.TT_evap / HeatBalance.t_evap : HeatBalance.Te);
        Ex.ExecuteLine (ExpertLine);
        ExpertLine = "TempTimeHot=";
        ExpertLine += (HeatBalance.t_hot ? HeatBalance.TT_hot / HeatBalance.t_hot : HeatBalance.Th);
        Ex.ExecuteLine (ExpertLine);
    }
    Ex.FinishExpert ();
}

//---------------------------------------------------------------------------
```

SwineSimSimulation.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimSimulationH
#define SwineSimSimulationH
```

```
//---------------------------------------------------------------------------

struct TSimulatedData {
    double Time;
    double Age;
    double Ta, Tr, Tf, phia, va, I;
       // air, radiant and floor temperatures (C), relative humidity (%),
       // air speed (m/s), solar radiation (W/m2)
    double Tb, Tsa, Tse, Tsf, Th, Te, Tc;
       // body, sunlight skin, shaded skin and floor skin temperatures (C),
       // upper, evaporative and lower critical temperatures (C)
    double qP, qL;
       // rate of heat production (W), rate of heat loss (W)
    double qss, qse, qsf, qv, ql;
       // heat loss acrosss the sunlight skin, shaded skin and floor skin (W),
       // heat loss through breathing (W), heat loss to ingested matter (W)
    double qsec, qser, qsee;
       // heat loss across the shaded skin through
       // convection, radiation and evaporation (W/m2)
    double qssc, qssr, qsse, qsss;
       // heat loss across the sunlight skin through
       // convection, radiation, evaporation and solar (W/m2)
    double qsm, qpq, qpd, qpl, qpf;
       // heat production for maintenance, additional heat, digestion,
       // lean tissue metabolism and fat tissue metabolism (W)
    double Np [POOL_NUTRIENTS]; // nutrient pool
    double Ni [FEED_NUTRIENTS]; // ingested nutrients
    double dWlt, dWft;
       // rate of lean and fat tissue deposition (kg/day)
    double Wb, We, Wlt, Wft;
       // body weight, equivalent body weight, lean tissue and fat tissue (kg)
    double DdWb, DGF;
       // weight gain (kg/day), feed consumption (kg/day) and gain:feed ratio in the last 24 h
    double DRa, DRc, DRw;
       // feed supplied, consumed and wasted (kg/day) in the last 24 h
    double Rf, Ra, Rl;
       // feed in feeder (kg), available feed (kg), feeding level (kg/day)
    int Phase; // feeding phase number
    double dWb, FL; // rate of weight gain (kg/day), fat to lean ratio
};

struct TDailyTotals {
    double Rs;
    double Rc;
    double Rw;
    double Wb;
};

class TSimulation {
    friend TDataTransfer;
public:
    int StartSimulation (void);
    int ContinueSimulation (void);
    void FinishSimulation (void);
    void SimulateStep (void);
    double GetRecordingTime (void) { return (RecordingTime / 24); };
private:
    // State variables (need to be initialized):
    int InitialAge;
    int CurrentAge;
    double InitialWeight;
    double CurrentWeight;
    double SimulationTime; // time elapsed from the beginning of the simulation
    double TimeOfDay; // current time of day
    double RecordingTime; // time elapsed from the last time the program stopped
    double UpdateGraphTime; // next time to update the simulation graph
    // Simulation parameters:
    int StartAge;
    double StartWeight;
    bool FinishOnDays;
    int SimulatedDays;
    double FinishWeight;
    double TimeStep;
    bool ConstantWeight;
    int GraphType;
    // Derived parameters (need to be initialized):
    int FinishAge;
    double DataLoggingTime; // time of next data logging event
    int LoggingIndex;
    static const double LoggingInterval [11];
    void RecordData (TSimulatedData & SimData);
    void RecordGraphData (bool NewData);
    void RecordDayData (void);
    void WriteSimulatedData (int Index);
};

extern TSimulatedData SimulatedData [STEPS_PER_DAY * 10 + 1];
```

```
extern TSimulatedData DailySimData [MAX_RECORDED_DAYS + 1];
extern TDailyTotals DailyTotals [STEPS_PER_DAY + 1];
extern TSimulation Simulation;
//---------------------------------------------------------------------------
#endif
```

SwineSimSimulation.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include <math.h>
#include "SwineSim.h"
#include "SwineSimSimulation.h"
//---------------------------------------------------------------------------
TSimulation Simulation;
TSimulatedData SimulatedData [STEPS_PER_DAY * 10 + 1];
TSimulatedData DailySimData [MAX_RECORDED_DAYS + 1];
TDailyTotals DailyTotals [STEPS_PER_DAY + 1];

const double TSimulation::LoggingInterval [11] = { 1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 1e99 };

void __fastcall TMainWindow::IdleEvent (System::TObject* Sender, bool &Done) {
    if (SimulationRunning && ! SimulationStopped) {
        try {
            Simulation.SimulateStep ();
        } catch (char * Message) {
            Application->MessageBox (Message, "Error in simulation", MB_OK);
            MainWindow->SimBtnStart->Caption = "Continue";
            MainWindow->SimBtnReset->Caption = "Reset";
            MainWindow->SimulationRunning = 0;
            MainWindow->SimulationStopped = 1;
            MainWindow->StatusBar->SimpleText = "Simulation stopped due to errors";
        }
        Done = false;
    } else
        Done = true;
}

// Reads all simulation variables from the disk file to memory:
void TDataTransfer::ReadSimulationFromFile (TIniFile * IniFile) {
    System::AnsiString Section = "Simulation";
    Simulation.StartAge = IniFile->ReadInteger (Section, "StartingAge", 60);
    Simulation.StartWeight = atof (IniFile->ReadString (Section, "StartingWeight", 20).c_str());
    Simulation.FinishOnDays = IniFile->ReadBool (Section, "FinishOnDays", true);
    Simulation.SimulatedDays = IniFile->ReadInteger(Section, "SimulatedDays", 2);
    Simulation.FinishWeight = atof (IniFile->ReadString (Section, "FinishingWeight", 100).c_str());
    Simulation.TimeStep = atof (IniFile->ReadString (Section, "TimeStep", 0.025).c_str());
    Simulation.ConstantWeight = IniFile->ReadBool (Section, "ConstantWeight", false);
    Simulation.GraphType = IniFile->ReadInteger (Section, "GraphType", 0);
    Simulation.TimeOfDay = 0;
    Simulation.SimulationTime = 0;
    Simulation.CurrentAge = Simulation.StartAge;
    Simulation.CurrentWeight = Simulation.StartWeight;
}

// Writes all simulation variables from memory to the disk file:
void TDataTransfer::WriteSimulationToFile (TIniFile * IniFile) {
    System::AnsiString Section = "Simulation";
    IniFile->WriteInteger (Section, "StartingAge", Simulation.StartAge);
    IniFile->WriteString (Section, "StartingWeight", Simulation.StartWeight);
    IniFile->WriteBool (Section, "FinishOnDays", Simulation.FinishOnDays);
    IniFile->WriteInteger (Section, "SimulatedDays", Simulation.SimulatedDays);
    IniFile->WriteString (Section, "FinishingWeight", Simulation.FinishWeight);
    IniFile->WriteString (Section, "TimeStep", Simulation.TimeStep);
    IniFile->WriteBool (Section, "ConstantWeight", Simulation.ConstantWeight);
    IniFile->WriteInteger (Section, "GraphType", Simulation.GraphType);
}

// Reads all simulation variables from the GUI to memory:
void TDataTransfer::ReadSimulationFromScreen (void) {
    Simulation.StartAge = atoi (MainWindow->SimStartAge->Text.c_str());
    Simulation.StartWeight = atof (MainWindow->SimStartWeight->Text.c_str());
    Simulation.FinishOnDays = MainWindow->SimRbFinishDays->Checked;
    Simulation.SimulatedDays = atoi (MainWindow->SimSimulatedDays->Text.c_str());
    Simulation.FinishWeight = atof (MainWindow->SimFinishWeight->Text.c_str());
    Simulation.TimeStep = atof (MainWindow->SimTimeStep->Text.c_str());
    Simulation.ConstantWeight = MainWindow->SimConstantWeight->Checked;
    Simulation.GraphType = MainWindow->SimGraphType->ItemIndex;
}
```

```
// Writes all simulation variables from memory to the GUI:
void TDataTransfer::WriteSimulationToScreen (void) {
    MainWindow->SimStartAge->Text = Simulation.StartAge;
    MainWindow->SimStartWeight->Text = Simulation.StartWeight;
    MainWindow->SimRbFinishDays->Checked = Simulation.FinishOnDays;
    MainWindow->SimSimulatedDays->Enabled = Simulation.FinishOnDays;
    MainWindow->SimDaysLabel->Enabled = Simulation.FinishOnDays;
    MainWindow->SimRbFinishWeight->Checked = ! Simulation.FinishOnDays;
    MainWindow->SimFinishWeight->Enabled = ! Simulation.FinishOnDays;
    MainWindow->SimKgLabel->Enabled = ! Simulation.FinishOnDays;
    MainWindow->SimSimulatedDays->Text = Simulation.SimulatedDays;
    MainWindow->SimFinishWeight->Text = Simulation.FinishWeight;
    MainWindow->SimTimeStep->Text = Simulation.TimeStep;
    MainWindow->SimConstantWeight->Checked = Simulation.ConstantWeight;
    MainWindow->SimGraphType->ItemIndex = Simulation.GraphType;
    MainWindow->SimTimeOfDay->Text = Simulation.TimeOfDay;
    MainWindow->SimSimulationDay->Text = (int) (Simulation.SimulationTime / 24);
    MainWindow->SimCurrentAge->Text = Simulation.CurrentAge;
    MainWindow->SimCurrentWeight->Text = Simulation.CurrentWeight;
}

// Updates the simulation graph:
void TMainWindow::UpdateSimGraph (void) {
    short i, j;
    double FinalAge = Simulation.GetRecordingTime();
    double PointAge;
    SimGraph->NumPoints = 2 * STEPS_PER_DAY + 1;
    switch (SimGraphType->ItemIndex) {
        case 0:
            SimGraph->GraphTitle = "Environmental variables";
            SimGraph->LeftTitle = "";
            SimGraph->NumSets = 6;
            SimGraph->ThisSet = 1;
                SimGraph->LegendText = "Air\nTemp (°C)";
                SimGraph->ColorData = gphCyan;
                SimGraph->PatternData = 2;
            SimGraph->ThisSet = 2;
                SimGraph->LegendText = "Radiant\nTemp (°C)";
                SimGraph->ColorData = gphRed;
                SimGraph->PatternData = 4;
            SimGraph->ThisSet = 3;
                SimGraph->LegendText = "Floor\nTemp (°C)";
                SimGraph->ColorData = gphBrown;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 4;
                SimGraph->LegendText = "Humidity\n(10 = 20 %)";
                SimGraph->ColorData = gphLightBlue;
                SimGraph->PatternData = 3;
            SimGraph->ThisSet = 5;
                SimGraph->LegendText = "Air Speed\n(10 = 1 m/s)";
                SimGraph->ColorData = gphGreen;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 6;
                SimGraph->LegendText = "Radiation\n(10 = 200 W/m²)";
                SimGraph->ColorData = gphLightRed;
                SimGraph->PatternData = 2;
        break;
        case 1:
            SimGraph->GraphTitle = "Body temperatures";
            SimGraph->LeftTitle = "°C";
            SimGraph->NumSets = 7;
            SimGraph->ThisSet = 1;
                SimGraph->LegendText = "Th";
                SimGraph->ColorData = gphLightGray;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 2;
                SimGraph->LegendText = "Te";
                SimGraph->ColorData = gphLightGray;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 3;
                SimGraph->LegendText = "Tc";
                SimGraph->ColorData = gphLightGray;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 4;
                SimGraph->LegendText = "Body";
                SimGraph->ColorData = gphGreen;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 5;
                SimGraph->LegendText = "Skin\n(sunlight)";
                SimGraph->ColorData = gphLightRed;
                SimGraph->PatternData = 2;
            SimGraph->ThisSet = 6;
                SimGraph->LegendText = "Skin\n(shaded)";
                SimGraph->ColorData = gphRed;
                SimGraph->PatternData = 4;
            SimGraph->ThisSet = 7;
```

```
        SimGraph->LegendText = "Skin\n(floor)";
        SimGraph->ColorData = gphBrown;
        SimGraph->PatternData = 0;
break;
case 2:
    SimGraph->GraphTitle = "Heat balance";
    SimGraph->LeftTitle = "W";
    SimGraph->NumSets = 2;
    SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Heat\nproduction";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Heat\nloss";
        SimGraph->ColorData = gphLightBlue;
        SimGraph->PatternData = 0;
break;
case 3:
    SimGraph->GraphTitle = "Heat loss";
    SimGraph->LeftTitle = "W";
    SimGraph->NumSets = 5;
    SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Shaded\nenvironment";
        SimGraph->ColorData = gphRed;
        SimGraph->PatternData = 4;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Sunlight\nenvironment";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Floor";
        SimGraph->ColorData = gphBrown;
        SimGraph->PatternData = 0;
    SimGraph->ThisSet = 4;
        SimGraph->LegendText = "Breathing";
        SimGraph->ColorData = gphLightBlue;
        SimGraph->PatternData = 3;
    SimGraph->ThisSet = 5;
        SimGraph->LegendText = "Ingested\nmatter";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 0;
break;
case 4:
    SimGraph->GraphTitle = "Skin heat loss rate (shaded)";
    SimGraph->LeftTitle = "W/m²";
    SimGraph->NumSets = 3;
    SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Convection";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 0;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Radiation";
        SimGraph->ColorData = gphRed;
        SimGraph->PatternData = 4;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Evaporation";
        SimGraph->ColorData = gphLightBlue;
        SimGraph->PatternData = 3;
break;
case 5:
    SimGraph->GraphTitle = "Skin heat loss rate (sunlight)";
    SimGraph->LeftTitle = "W/m²";
    SimGraph->NumSets = 4;
    SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Convection";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 0;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Radiation";
        SimGraph->ColorData = gphRed;
        SimGraph->PatternData = 4;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Evaporation";
        SimGraph->ColorData = gphLightBlue;
        SimGraph->PatternData = 3;
    SimGraph->ThisSet = 4;
        SimGraph->LegendText = "Solar";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
break;
case 6:
    SimGraph->GraphTitle = "Heat production";
    SimGraph->LeftTitle = "W";
    SimGraph->NumSets = 5;
    SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Maintenance";
        SimGraph->ColorData = gphMagenta;
```

```
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Additional\nheat";
        SimGraph->ColorData = gphLightBlue;
        SimGraph->PatternData = 3;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Digestion";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 0;
    SimGraph->ThisSet = 4;
        SimGraph->LegendText = "Lean tissue";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 5;
        SimGraph->LegendText = "Fat tissue";
        SimGraph->ColorData = gphBrown;
        SimGraph->PatternData = 0;
break;
case 7:
   SimGraph->GraphTitle = "Nutrient pool";
   SimGraph->LeftTitle = "g";
   SimGraph->NumSets = 3;
   SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Carbohydrates";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 3;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Protein";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Fat";
        SimGraph->ColorData = gphBrown;
        SimGraph->PatternData = 0;
break;
case 8:
   SimGraph->GraphTitle = "Ingested nutrients";
   SimGraph->LeftTitle = "g";
   SimGraph->NumSets = 4;
   SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Carbohydrates";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 3;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Protein";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Fat";
        SimGraph->ColorData = gphBrown;
        SimGraph->PatternData = 0;
    SimGraph->ThisSet = 4;
        SimGraph->LegendText = "Fiber";
        SimGraph->ColorData = gphLightMagenta;
        SimGraph->PatternData = 2;
break;
case 9:
   SimGraph->GraphTitle = "Tissue growth rates";
   SimGraph->LeftTitle = "kg/day";
   SimGraph->NumSets = 2;
   SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Lean";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Fat";
        SimGraph->ColorData = gphBrown;
        SimGraph->PatternData = 0;
break;
case 10:
   SimGraph->GraphTitle = "Growth curves";
   SimGraph->LeftTitle = "kg";
   SimGraph->NumSets = 4;
   SimGraph->ThisSet = 1;
        SimGraph->LegendText = "Body weight";
        SimGraph->ColorData = gphCyan;
        SimGraph->PatternData = 0;
    SimGraph->ThisSet = 2;
        SimGraph->LegendText = "Equivalent\nbody weight";
        SimGraph->ColorData = gphLightBlue;
        SimGraph->PatternData = 3;
    SimGraph->ThisSet = 3;
        SimGraph->LegendText = "Lean tissue";
        SimGraph->ColorData = gphLightRed;
        SimGraph->PatternData = 2;
    SimGraph->ThisSet = 4;
        SimGraph->LegendText = "Fat tissue";
```

```
            SimGraph->ColorData = gphBrown;
            SimGraph->PatternData = 0;
        break;
        case 11:
            SimGraph->GraphTitle = "Feed consumption and daily gain";
            SimGraph->LeftTitle = "";
            SimGraph->NumSets = 3;
            SimGraph->ThisSet = 1;
                SimGraph->LegendText = "Feed intake\n(kg/day)";
                SimGraph->ColorData = gphLightBlue;
                SimGraph->PatternData = 3;
            SimGraph->ThisSet = 2;
                SimGraph->LegendText = "Weight gain\n(kg/day)";
                SimGraph->ColorData = gphCyan;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 3;
                SimGraph->LegendText = "Gain:feed";
                SimGraph->ColorData = gphLightRed;
                SimGraph->PatternData = 2;
        break;
        case 12:
            SimGraph->GraphTitle = "Feed usage";
            SimGraph->LeftTitle = "kg/day";
            SimGraph->NumSets = 3;
            SimGraph->ThisSet = 1;
                SimGraph->LegendText = "Feed\nsupplied";
                SimGraph->ColorData = gphGreen;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 2;
                SimGraph->LegendText = "Feed\nconsumed";
                SimGraph->ColorData = gphLightBlue;
                SimGraph->PatternData = 3;
            SimGraph->ThisSet = 3;
                SimGraph->LegendText = "Feed\nwasted";
                SimGraph->ColorData = gphLightRed;
                SimGraph->PatternData = 2;
        break;
        case 13:
            SimGraph->GraphTitle = "Available feed";
            SimGraph->LeftTitle = "kg";
            SimGraph->NumSets = 3;
            SimGraph->ThisSet = 1;
                SimGraph->LegendText = "Feed in\nfeeder";
                SimGraph->ColorData = gphGreen;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 2;
                SimGraph->LegendText = "Available\nfeed";
                SimGraph->ColorData = gphLightBlue;
                SimGraph->PatternData = 3;
            SimGraph->ThisSet = 3;
                SimGraph->LegendText = "Feeding\nlevel";
                SimGraph->ColorData = gphBrown;
                SimGraph->PatternData = 2;
        break;
        case 14:
            SimGraph->GraphTitle = "Feeding phases";
            SimGraph->LeftTitle = "";
            SimGraph->NumSets = 2;
            SimGraph->ThisSet = 1;
                SimGraph->LegendText = "Body weight\n(kg)";
                SimGraph->ColorData = gphCyan;
                SimGraph->PatternData = 0;
            SimGraph->ThisSet = 2;
                SimGraph->LegendText = "Phase";
                SimGraph->ColorData = gphDarkGray;
                SimGraph->PatternData = 2;
        break;
        default:
            SimGraph->NumSets = 1;
        break;
    }
    for (i = 0; i <= 2 * STEPS_PER_DAY; i ++) {
        SimGraph->ThisPoint = (short) (i + 1);
        PointAge = i * FinalAge / (2 * STEPS_PER_DAY) + ROUND_OFF;
        if (PointAge < 2)
            j = ((PointAge + 0) * STEPS_PER_DAY) / 1;
        else if (PointAge < 4)
            j = ((PointAge + 2)* STEPS_PER_DAY) / 2;
        else if (PointAge < 8)
            j = ((PointAge + 8) * STEPS_PER_DAY) / 4;
        else if (PointAge < 16)
            j = ((PointAge + 24) * STEPS_PER_DAY) / 8;
        else if (PointAge < 32)
            j = ((PointAge + 64) * STEPS_PER_DAY) / 16;
        else if (PointAge < 64)
            j = ((PointAge + 160) * STEPS_PER_DAY) / 32;
        else if (PointAge < 128)
```

```
    j = ((PointAge + 384) * STEPS_PER_DAY) / 64;
else if (PointAge < 256)
    j = ((PointAge + 896) * STEPS_PER_DAY) / 128;
else if (PointAge < 512)
    j = ((PointAge + 2048) * STEPS_PER_DAY) / 256;
else
    j = STEPS_PER_DAY * 10;
PointAge -= 2 * ROUND_OFF;
if (PointAge < 0) PointAge = 0;
switch (SimGraphType->ItemIndex) {
    case 0:
        SimGraph->ThisSet = 1;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Ta;
        SimGraph->ThisSet = 2;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tr;
        SimGraph->ThisSet = 3;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tf;
        SimGraph->ThisSet = 4;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].phia / 2;
        SimGraph->ThisSet = 5;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].va * 10;
        SimGraph->ThisSet = 6;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].I / 20;
    break;
    case 1:
        SimGraph->ThisSet = 1;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Th;
        SimGraph->ThisSet = 2;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Te;
        SimGraph->ThisSet = 3;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tc;
        SimGraph->ThisSet = 4;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tb;
        SimGraph->ThisSet = 5;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tss;
        SimGraph->ThisSet = 6;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tse;
        SimGraph->ThisSet = 7;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].Tsf;
    break;
    case 2:
        SimGraph->ThisSet = 1;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qP;
        SimGraph->ThisSet = 2;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qL;
    break;
    case 3:
        SimGraph->ThisSet = 1;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qse;
        SimGraph->ThisSet = 2;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qss;
        SimGraph->ThisSet = 3;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qsf;
        SimGraph->ThisSet = 4;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qv;
        SimGraph->ThisSet = 5;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qi;
    break;
    case 4:
        SimGraph->ThisSet = 1;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qsec;
        SimGraph->ThisSet = 2;
            SimGraph->XPosData = PointAge;
            SimGraph->GraphData = SimulatedData[j].qser;
        SimGraph->ThisSet = 3;
            SimGraph->XPosData = PointAge;
```

```
        SimGraph->GraphData = SimulatedData[j].qsee;
break;
case 5:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qssc;
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qssr;
    SimGraph->ThisSet = 3;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qsse;
    SimGraph->ThisSet = 4;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qsss;
break;
case 6:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qpm;
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qpq;
    SimGraph->ThisSet = 3;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qpd;
    SimGraph->ThisSet = 4;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qpl;
    SimGraph->ThisSet = 5;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].qpf;
break;
case 7:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Np[POOL_CARBS];
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Np[POOL_PROTEIN];
    SimGraph->ThisSet = 3;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Np[POOL_FAT];
break;
case 8:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Ni[FEED_CARBS];
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Ni[FEED_PROTEIN];
    SimGraph->ThisSet = 3;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Ni[FEED_FAT];
    SimGraph->ThisSet = 4;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Ni[FEED_FIBER];
break;
case 9:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].dWlt;
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].dWft;
break;
case 10:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Wb;
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].We;
    SimGraph->ThisSet = 3;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Wlt;
    SimGraph->ThisSet = 4;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].Wft;
break;
case 11:
    SimGraph->ThisSet = 1;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].DRc;
    SimGraph->ThisSet = 2;
        SimGraph->XPosData = PointAge;
        SimGraph->GraphData = SimulatedData[j].DdWb;
```

```
            SimGraph->ThisSet = 3;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].DGF;
        break;
        case 12:
            SimGraph->ThisSet = 1;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].DRs + ROUND_OFF;
            SimGraph->ThisSet = 2;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].DRc + ROUND_OFF;
            SimGraph->ThisSet = 3;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].DRw + ROUND_OFF;
        break;
        case 13:
            SimGraph->ThisSet = 1;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].Rf + ROUND_OFF;
            SimGraph->ThisSet = 2;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].Ra + ROUND_OFF;
            SimGraph->ThisSet = 3;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].R1 + ROUND_OFF;
        break;
        case 14:
            SimGraph->ThisSet = 1;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].Wb;
            SimGraph->ThisSet = 2;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimulatedData[j].Phase * 10;
        break;
        default:
            SimGraph->ThisSet = 1;
                SimGraph->XPosData = PointAge;
                SimGraph->GraphData = SimGraphType->ItemIndex;
        break;
      }
    }
    SimGraph->DrawMode = gphBlit;
}


void TSimulation::RecordData (TSimulatedData & SimData) {
    int j;
    SimData.Time = RecordingTime;
    SimData.Age = CurrentAge;
    // Graph 0:
    SimData.Ta = Environment.Ta;
    SimData.Tr = Environment.Tr;
    SimData.Tf = Environment.Tf;
    SimData.phia = Environment.phia;
    SimData.va = Environment.va;
    SimData.I = Environment.I;
    // Graph 1:
    SimData.Tb = HeatBalance.Tb;
    SimData.Tss = HeatBalance.Tss;
    SimData.Tse = HeatBalance.Tse;
    SimData.Tsf = HeatBalance.Tsf;
    SimData.Th = HeatBalance.Th;
    SimData.Te = HeatBalance.Te;
    SimData.Tc = HeatBalance.Tc;
    // Graph 2:
    SimData.qP = Metabolism.QP / (3.6 * TimeStep);
    SimData.qL = HeatBalance.QL / (3.6 * TimeStep);
    // Graph 3:
    SimData.qss = HeatBalance.qRs * HeatBalance.ks * HeatBalance.Ab;
    SimData.qse = HeatBalance.qRe * HeatBalance.ke * HeatBalance.Ab;
    SimData.qsf = HeatBalance.qRf * HeatBalance.kf * HeatBalance.Ab;
    SimData.qv = HeatBalance.qv;
    SimData.qi = HeatBalance.qi;
    // Graph 4:
    SimData.qsec = HeatBalance.qRe_conv;
    SimData.qser = HeatBalance.qRe_rad;
    SimData.qsee = HeatBalance.qRe_evap;
    // Graph 5:
    SimData.qssc = HeatBalance.qRs_conv;
    SimData.qssr = HeatBalance.qRs_rad;
    SimData.qsse = HeatBalance.qRs_evap;
    SimData.qsss = HeatBalance.qRs_sol;
    // Graph 6:
    SimData.qpm = Metabolism.QP_maintenance / (3.6 * TimeStep);
    SimData.qpq = Metabolism.QP_heat / (3.6 * TimeStep);
    SimData.qpd = Metabolism.QP_digestion / (3.6 * TimeStep);
    SimData.qpl = Metabolism.QP_lean / (3.6 * TimeStep);
```

```
    SimData.qpf = Metabolism.QP_fat / (3.6 * TimeStep);
    // Graph 7:
    for (j = 0; j < POOL_NUTRIENTS; j ++)
      SimData.Np[j] = Metabolism.Np[j];
    // Graph 8:
    for (j = 0; j < FEED_NUTRIENTS; j ++)
      SimData.Ni[j] = Metabolism.Ni[j];
    // Graph 9:
    SimData.dWlt = Metabolism.dWlt;
    SimData.dWft = Metabolism.dWft;
    // Graph 10:
    SimData.Wb = Metabolism.Wb;
    SimData.We = Metabolism.We;
    SimData.Wlt = Metabolism.Wlt;
    SimData.Wft = Metabolism.Wft;
    // Graph 11:
    SimData.DRc = DailyTotals[0].Rc;
    SimData.DdWb = DailyTotals[0].Wb;
    SimData.DGF = (DailyTotals[0].Rc > DailyTotals[0].Wb &&
      DailyTotals[0].Wb > 0 ? DailyTotals[0].Wb / DailyTotals[0].Rc : 0);
    // Graph 12:
    SimData.DRs = DailyTotals[0].Rs;
    SimData.DRw = DailyTotals[0].Rw;
    // Graph 13:
    SimData.Rf = FeedingProgram.FeedInFeeder;
    SimData.Ra = FeedingProgram.AvailableFeed;
    SimData.Rl = FeedingProgram.FeedingAmount;
    // Graph 14:
    SimData.Phase = FeedingProgram.ThisPhase + 1;
    // Other variables:
    SimData.dWb = (Metabolism.dWlt*(1+Metabolism.kWT) + Metabolism.dWft)/Metabolism.kEB;
    SimData.FL = (Metabolism.Wlt ? Metabolism.Wft / Metabolism.Wlt : 0);
}

void TSimulation::RecordDayData (void) {
    RecordData [DailySimData [CurrentAge - InitialAge > MAX_RECORDED_DAYS ?
        MAX_RECORDED_DAYS : CurrentAge - InitialAge]);
}

void TSimulation::RecordGraphData (bool NewData) {
    int DailyStep =((int) (STEPS_PER_DAY * TimeOfDay / 24 + ROUND_OFF)) % STEPS_PER_DAY + 1;
    if (NewData) {
        DailyTotals[0].Rs = FeedingProgram.FeedSupplied-DailyTotals[DailyStep].Rs;
        DailyTotals[0].Rc = FeedingProgram.FeedConsumed-DailyTotals[DailyStep].Rc;
        DailyTotals[0].Rw = FeedingProgram.FeedWasted - DailyTotals[DailyStep].Rw;
        DailyTotals[0].Wb = Metabolism.Wb - DailyTotals[DailyStep].Wb;
        if (STEPS_PER_DAY * TimeOfDay / 24 - DailyStep + 1 < ROUND_OFF) {
            DailyTotals[DailyStep].Rs = FeedingProgram.FeedSupplied;
            DailyTotals[DailyStep].Rc = FeedingProgram.FeedConsumed;
            DailyTotals[DailyStep].Rw = FeedingProgram.FeedWasted;
            DailyTotals[DailyStep].Wb = Metabolism.Wb;
        }
    }
    while (RecordingTime + ROUND_OFF > DataLoggingTime) {
        RecordData [SimulatedData[LoggingIndex]];
        // Set the next time to record data:
        DataLoggingTime += (24.0 / STEPS_PER_DAY) *
            LoggingInterval [LoggingIndex / STEPS_PER_DAY];
        LoggingIndex ++;
    }
}

void TSimulation::SimulateStep (void) {
    // Test if the simulation is finished:
    if (FinishOnDays ? CurrentAge >= FinishAge : CurrentWeight >= FinishWeight) {
        FinishSimulation ();
        return;
    }
    // Update current times and check for a change of date:
    if (TimeStep <= 0)
        throw ("Time Step is zero or negative");
    bool DayChange = false;
    SimulationTime += TimeStep;
    RecordingTime += TimeStep;
    TimeOfDay += TimeStep;
    if (TimeOfDay + ROUND_OFF >= 24) {
        DayChange = true;
        CurrentAge ++;
        TimeOfDay -= 24;
        MainWindow->SimCurrentAge->Text = CurrentAge;
    }
    // Execute the time step:
    Environment.SimulateStep (TimeOfDay);
    FeedingProgram.SimulateStep (TimeOfDay, DayChange, CurrentAge, CurrentWeight);
    Metabolism.SimulateStep (TimeStep, CurrentAge, ConstantWeight);
    HeatBalance.SimulateStep (TimeStep);
    FeedingProgram.RegisterFeedConsumption (Metabolism.Get_Ri());
```

```
    // Update the screen variables:
    CurrentWeight = Metabolism.Get_Wb ();
    MainWindow->SimCurrentWeight->Text =
       AnsiString::FloatToStrF(CurrentWeight, AnsiString::sffFixed, 4, 2);
    MainWindow->SimSimulationDay->Text =
       AnsiString::FloatToStrF((SimulationTime + ROUND_OFF) / 24, AnsiString::sffFixed, 4, 2);
    MainWindow->SimTimeOfDay->Text =
       AnsiString::FloatToStrF(TimeOfDay + ROUND_OFF, AnsiString::sffFixed, 4, 2);
    // Log data:
    RecordGraphData(true);
    if (DayChange)
       RecordDayData ();
    // Update the graph every day for the first 10 days, every 5 days for the
    // first 50 days, every 10 days for the first 200 days, every 20 days from
    // there on:
    if (RecordingTime + ROUND_OFF > UpdateGraphTime) {
       MainWindow->UpdateSimGraph ();
       UpdateGraphTime += (RecordingTime + ROUND_OFF < 240 ? 24 :
          (RecordingTime + ROUND_OFF < 1200 ? 120 :
          (RecordingTime + ROUND_OFF < 4800 ? 240 : 480)));
    }
}

int TSimulation::StartSimulation (void) {
    MainWindow->StatusBar->SimpleText = "Initializing simulation";
    DataTransfer.ReadSimulationFromScreen ();
    DataTransfer.ReadEnvironmentFromScreen ();
    DataTransfer.ReadFeedingProgramFromScreen ();
    DataTransfer.ReadHeatBalanceFromScreen ();
    DataTransfer.ReadMetabolismFromScreen ();
    CurrentAge = InitialAge = StartAge;
    CurrentWeight = InitialWeight = StartWeight;
    SimulationTime = 0;
    TimeOfDay = 0;
    try {
       Environment.SimulateStep (TimeOfDay);
       FeedingProgram.StartSimulation (TimeOfDay, CurrentAge, CurrentWeight);
       Metabolism.StartSimulation (CurrentWeight, CurrentAge, TimeStep);
       HeatBalance.StartSimulation (TimeStep);
    } catch (char * ErrorMessage) {
       MainWindow->StatusBar->SimpleText = "Simulation error; please fix it!";
       Application->MessageBox (ErrorMessage, "Data error", MB_OK);
       MainWindow->StatusBar->SimpleText = "Simulation did not start due to errors";
       return (0);
    }
    memset (DailyTotals, 0, sizeof (DailyTotals));
    memset (DailySimData, 0, sizeof (DailySimData));
    for (int i = 1; i <= STEPS_PER_DAY; i ++)
       DailyTotals[i].Wb = StartWeight;
    if (ContinueSimulation()) {
       RecordDayData ();
       return (1);
    } else
       return (0);
}

int TSimulation::ContinueSimulation (void) {
    MainWindow->StatusBar->SimpleText = "Starting simulation";
    DataTransfer.ReadSimulationFromScreen ();
    DataTransfer.ReadEnvironmentFromScreen ();
    DataTransfer.ReadFeedingProgramFromScreen ();
    DataTransfer.ReadHeatBalanceFromScreen ();
    DataTransfer.ReadMetabolismFromScreen ();
    RecordingTime = 0;
    DataLoggingTime = 0;
    UpdateGraphTime = 0;
    FinishAge = CurrentAge + SimulatedDays;
    memset (SimulatedData, 0, sizeof (SimulatedData));
    LoggingIndex = 0;
    try {
       Environment.SimulateStep (TimeOfDay);
       FeedingProgram.ContinueSimulation (TimeOfDay, CurrentAge, CurrentWeight);
    } catch (char * ErrorMessage) {
       MainWindow->StatusBar->SimpleText = "Simulation error; please fix it!";
       Application->MessageBox (ErrorMessage, "Data error", MB_OK);
       MainWindow->StatusBar->SimpleText = "Simulation did not start due to errors";
       return (0);
    }
    RecordGraphData(false);
    MainWindow->SimCurrentAge->Text = CurrentAge;
    MainWindow->StatusBar->SimpleText = "Simulation started";
    return (1);
}

void TSimulation::FinishSimulation (void) {
    MainWindow->UpdateSimGraph ();
    DataTransfer.ExecuteExpertSystem ();
```

```
    DataTransfer.SaveSimulationFile ();
    MainWindow->SimBtnStart->Caption = "Continue";
    MainWindow->SimBtnReset->Caption = "Reset";
    MainWindow->SimulationRunning = 0;
    MainWindow->SimulationStopped = 1;
    MainWindow->StatusBar->SimpleText = "Simulation finished";
}
```

SwineSimEnvironment.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimEnvironmentH
#define SwineSimEnvironmentH
//---------------------------------------------------------------------------

#include <math.h>

// This class keeps the values of the simulation variables related to the
// microenvironment module of the simulation model:
class TEnvironment {
    friend TDataTransfer;
    friend void TSimulation::RecordData (TSimulatedData & SimData);
public:
    void StartSimulation (void);
    void SimulateStep (double TimeOfDay);
    double Get_Ta (void) { return (Ta); };
    double Get_Tf (void) { return (Tf); };
    double Get_Tr (void) { return (Tr); };
    double Get_phia (void) { return (phia / 100); };
    double Get_va (void) { return (va); };
    double Get_I (void) { return (I); };
    double Get_Ti (void) { return (FeedTempEqualAir ? Ta : Ti); };
    double Get_Tw (void) { return (Tw); };
    double SaturationVaporPressure (double T) {
        // Saturation vapor pressure as a function of temperature (Pa)
        T = T + 273.15;
        if (T < 173.15)
            T = 173.15;
        if (T < 273.15)
            return (exp (- 5674.5359 / T + 6.3925247 - 0.009677843 * T + 6.22115701e-7 *          T*T +
                2.0747825e-9 * T*T*T - 9.484024e-13 * T*T*T*T + 4.1635019 * log (T)));
        if (T > 473.15)
            T = 473.15;
        return (exp (- 5800.2206 / T + 1.3914993 - 0.048640239 * T +
            4.1764768e-5 * T*T - 1.4452093e-8 * T*T*T + 6.5459673 * log (T)));
    };
    double EnthalpyOfAir (double T, double phi) {
        // Enthalpy of air as a function of temperature and relative humidity
        // at atmospheric pressure (kJ / kg)
        double pw = phi * SaturationVaporPressure (T);
        return (T + (2501 + 1.805 * T) * 0.62198 * pw / (1000*pa - pw));
    };
    double DensityOfAir (double T, double phi) {
        // Density of air as a function of temperature and relative humidity
        // at atmospheric pressure (kg / m3)
        double pw = phi * SaturationVaporPressure (T);
        return (1000*pa / (287.055 * (T + 273.15) * (1 + 1.6078 * (0.62198 * pw / (1000*pa - pw)))));
    };
private:
    double Ta; // Air temperature (°C)
    double Ta_min, Ta_max; // air temperature extremes
    double tTa_min, tTa_max; // time of air temperature extremes
    double Tf; // Temperature of the floor (°C)
    double Tf_min, Tf_max; // floor temperature extremes
    double tTf_min, tTf_max; // time of floor temperature extremes
    double Tr; // Temperature of the surrounding radiating surfaces (°C)
    double Tr_min, Tr_max; // radiant temperature extremes
    double tTr_min, tTr_max; // time of radiant temperature extremes
    double va; // Air speed (m/s)
    double van; // Natural air speed (m/s)
    double va_min1, va_max1, va_min2, va_max2; // air speed extremes
    double tVa_min1, tVa_max1, tVa_min2, tVa_max2; // time of air speed extremes
    double vaf; // Forced air speed (m/s)
    double tVa_on, tVa_off; // time when fans are turned on and off
    double phia; // Relative humidity of ambient air
    double phia_max, phia_min; // relative humidity extremes
    double tphia_max, tphia_min; // time of relative humidity extremes
    double I; // Intensity of solar radiation on a normal surface (W/m2)
    double I6; // Intensity of solar radiation on a normal surface (W/m2) at 6h
    double I7; // Intensity of solar radiation on a normal surface (W/m2) at 7h
    double I8; // Intensity of solar radiation on a normal surface (W/m2) at 8h
```

```
    double I9; // Intensity of solar radiation on a normal surface (W/m2) at 9h
    double I10; // Intensity of solar radiation on a normal surface (W/m2) at 10h
    double I11; // Intensity of solar radiation on a normal surface (W/m2) at 11h
    double I12; // Intensity of solar radiation on a normal surface (W/m2) at 12h
    bool FeedTempEqualAir; // true if feed temperature should be equal to air
    double Tf; // Temperature of ingested feed (°C)
    double Tw; // Temperature of ingested water (°C)
    double pa; // Ambient air pressure (kPa)
};

// This variable implements the TEnvironment class:
extern TEnvironment Environment;

//---------------------------------------------------------------------------
#endif
```

SwineSimEnvironment.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include <math.h>
#include "SwineSim.h"
#include "SwineSimEnvironment.h"
//---------------------------------------------------------------------------

// This variable implements the TEnvironment class:
TEnvironment Environment;

// Description of all variables that are transfered between disk file,
// memory and GUI:
TVariableDescription TDataTransfer::EnvironmentDescription[38];

#define SED(i,ADDRESS,EDIT,NAME,DEFAULT) {       \
    EnvironmentDescription[i].Address = (ADDRESS); \
    EnvironmentDescription[i].Edit = (EDIT);     \
    EnvironmentDescription[i].Name = (NAME);     \
    EnvironmentDescription[i].Default = (DEFAULT); \
}

void TDataTransfer::SetEnvironmentDescription (void) {
    SED (0, &(Environment.Ta_min), MainWindow->Ta_min, "MinimumAirTemperature", 15);
    SED (1, &(Environment.Ta_max), MainWindow->Ta_max, "MaximumAirTemperature", 25);
    SED (2, &(Environment.tTa_min), MainWindow->tTa_min, "TimeOfMinimumAirTemperature", 6);
    SED (3, &(Environment.tTa_max), MainWindow->tTa_max, "TimeOfMaximumAirTemperature", 15);
    SED (4, &(Environment.Tr_min), MainWindow->Tr_min, "MinimumRadiantTemperature", 15);
    SED (5, &(Environment.Tr_max), MainWindow->Tr_max, "MaximumRadiantTemperature", 25);
    SED (6, &(Environment.tTr_min), MainWindow->tTr_min, "TimeOfMinimumRadiantTemperature", 6);
    SED (7, &(Environment.tTr_max), MainWindow->tTr_max, "TimeOfMaximumRadiantTemperature", 15);
    SED (8, &(Environment.Tf_min), MainWindow->Tf_min, "MinimumFloorTemperature", 15);
    SED (9, &(Environment.Tf_max), MainWindow->Tf_max, "MaximumFloorTemperature", 25);
    SED (10, &(Environment.tTf_min), MainWindow->tTf_min, "TimeOfMinimumFloorTemperature", 6);
    SED (11, &(Environment.tTf_max), MainWindow->tTf_max, "TimeOfMaximumFloorTemperature", 15);
    SED (12, &(Environment.va_min1), MainWindow->va_min1, "MinimumMorningAirSpeed", 0);
    SED (13, &(Environment.va_max1), MainWindow->va_max1, "MaximumDaytimeAirSpeed", 2);
    SED (14, &(Environment.va_min2), MainWindow->va_min2, "MinimumAfternoonAirSpeed", 0);
    SED (15, &(Environment.va_max2), MainWindow->va_max2, "MaximumEveningAirSpeed", 2);
    SED (16, &(Environment.tVa_min1), MainWindow->tVa_min1, "TimeOfMinimumMorningAirSpeed", 6);
    SED (17, &(Environment.tVa_max1), MainWindow->tVa_max1, "TimeOfMaximumDaytimeAirSpeed", 10);
    SED (18, &(Environment.tVa_min2), MainWindow->tVa_min2, "TimeOfMinimumAfternoonAirSpeed", 15);
    SED (19, &(Environment.tVa_max2), MainWindow->tVa_max2, "TimeOfMaximumEveningAirSpeed", 20);
    SED (20, &(Environment.vaf), MainWindow->vaf, "ForcedAirSpeed", 3);
    SED (21, &(Environment.tVa_on), MainWindow->tVa_on, "TimeFansAreTurnedOn", 12);
    SED (22, &(Environment.tVa_off), MainWindow->tVa_off, "TimeFansAreTurnedOff", 18);
    SED (23, &(Environment.phia_max), MainWindow->phia_max, "MaximumRelativeHumidity", 100);
    SED (24, &(Environment.phia_min), MainWindow->phia_min, "MinimumRelativeHumidity", 70);
    SED (25, &(Environment.tphia_max), MainWindow->tphia_max, "TimeOfMaximumRelativeHumidity", 6);
    SED (26, &(Environment.tphia_min), MainWindow->tphia_min, "TimeOfMinimumRelativeHumidity", 15);
    SED (27, &(Environment.I6), MainWindow->I6, "SolarRadiationIntensity6h", 187);
    SED (28, &(Environment.I7), MainWindow->I7, "SolarRadiationIntensity7h", 599);
    SED (29, &(Environment.I8), MainWindow->I8, "SolarRadiationIntensity8h", 756);
    SED (30, &(Environment.I9), MainWindow->I9, "SolarRadiationIntensity9h", 830);
    SED (31, &(Environment.I10), MainWindow->I10, "SolarRadiationIntensity10h", 869);
    SED (32, &(Environment.I11), MainWindow->I11, "SolarRadiationIntensity11h", 888);
    SED (33, &(Environment.I12), MainWindow->I12, "SolarRadiationIntensity12h", 894);
    SED (34, &(Environment.Tf), MainWindow->Tf, "TemperatureOfFeed", 20);
    SED (35, &(Environment.Tw), MainWindow->Tw, "TemperatureOfWater", 20);
    SED (36, &(Environment.pa), MainWindow->pa, "AmbientAirPressure", 101.325);
    SED (37, NULL, NULL, NULL, 0);
}
```

```
// Initializes other GUI parameters related to environmental variables:
void TMainWindow::InitEnvTab (void) {
    EnvGraph->NumPoints = ENVGRAPH_DATAPOINTS + 1;
    EnvGraph->LabelEvery = ENVGRAPH_DATAPOINTS / 12;
    EnvGraph->TickEvery = ENVGRAPH_DATAPOINTS / 24;
    EnvGraph->GraphTitle = "Environmental variables";
    EnvGraph->LeftTitle = "";
    EnvGraph->NumSets = 6;
    EnvGraph->ThisSet = 1;
    EnvGraph->LegendText = "Air\nTemp (°C)";
    EnvGraph->ColorData = gphCyan; EnvGraph->PatternData = 2;
    EnvGraph->ThisSet = 2;
    EnvGraph->LegendText = "Radiant\nTemp (°C)";
    EnvGraph->ColorData = gphRed; EnvGraph->PatternData = 4;
    EnvGraph->ThisSet = 3;
    EnvGraph->LegendText = "Floor\nTemp (°C)";
    EnvGraph->ColorData = gphBrown; EnvGraph->PatternData = 0;
    EnvGraph->ThisSet = 4;
    EnvGraph->LegendText = "Humidity\n(10 = 20 %)";
    EnvGraph->ColorData = gphLightBlue; EnvGraph->PatternData = 3;
    EnvGraph->ThisSet = 5;
    EnvGraph->LegendText = "Air Speed\n(10 = 1 m/s)";
    EnvGraph->ColorData = gphGreen; EnvGraph->PatternData = 0;
    EnvGraph->ThisSet = 6;
    EnvGraph->LegendText = "Radiation\n(10 = 200 W/m²)";
    EnvGraph->ColorData = gphLightRed; EnvGraph->PatternData = 2;
    for (short i = 1; i <= ENVGRAPH_DATAPOINTS + 1;
        i += (short) (ENVGRAPH_DATAPOINTS / 12)) {
        EnvGraph->ThisPoint = i;
        EnvGraph->LabelText = (i - 1) * 24 / ENVGRAPH_DATAPOINTS;
    }
    DataTransfer.SetEnvironmentDescription ();
}

// Reads all environment variables from the disk file to memory:
void TDataTransfer::ReadEnvironmentFromFile (TIniFile * IniFile) {
    TVariableDescription * VarPtr;
    for (VarPtr = EnvironmentDescription; VarPtr->Address; VarPtr ++)
        *(VarPtr->Address) = atof (IniFile->ReadString ("Environment",
            VarPtr->Name, VarPtr->Default).c_str());
    Environment.FeedTempEqualAir = IniFile->ReadBool ("Environment", "FeedTempEqualAir", true);
}

// Writes all environment variables from memory to the disk file:
void TDataTransfer::WriteEnvironmentToFile (TIniFile * IniFile) {
    TVariableDescription * VarPtr;
    for (VarPtr = EnvironmentDescription; VarPtr->Address; VarPtr ++)
        IniFile->WriteString ("Environment", VarPtr->Name, *(VarPtr->Address));
    IniFile->WriteBool ("Environment", "FeedTempEqualAir", Environment.FeedTempEqualAir);
}

// Reads all environment variables from the GUI to memory:
void TDataTransfer::ReadEnvironmentFromScreen (void) {
    TVariableDescription * VarPtr;
    for (VarPtr = EnvironmentDescription; VarPtr->Address; VarPtr ++)
        *(VarPtr->Address) = atof (VarPtr->Edit->Text.c_str());
    Environment.FeedTempEqualAir = MainWindow->EnvFeedTempEqualAir->ItemIndex;
}

// Writes all environment variables from memory to the GUI:
void TDataTransfer::WriteEnvironmentToScreen (void) {
    TVariableDescription * VarPtr;
    for (VarPtr = EnvironmentDescription; VarPtr->Address; VarPtr ++)
        VarPtr->Edit->Text = *(VarPtr->Address);
    MainWindow->UpdateEnvGraph ();
    MainWindow->EnvFeedTempEqualAir->ItemIndex = Environment.FeedTempEqualAir;
    MainWindow->Fi->Enabled = ! MainWindow->EnvFeedTempEqualAir->ItemIndex;
}

// Upgates the graph of the environmental variables:
void TMainWindow::UpdateEnvGraph (void) {
    double Ta_min = atof (TMainWindow::Ta_min->Text.c_str());
    double tTa_min = atof (TMainWindow::tTa_min->Text.c_str());
    double Ta_max = atof (TMainWindow::Ta_max->Text.c_str());
    double tTa_max = atof (TMainWindow::tTa_max->Text.c_str());
    double Tr_min = atof (TMainWindow::Tr_min->Text.c_str());
    double tTr_min = atof (TMainWindow::tTr_min->Text.c_str());
    double Tr_max = atof (TMainWindow::Tr_max->Text.c_str());
    double tTr_max = atof (TMainWindow::tTr_max->Text.c_str());
    double Tf_min = atof (TMainWindow::Tf_min->Text.c_str());
    double tTf_min = atof (TMainWindow::tTf_min->Text.c_str());
    double Tf_max = atof (TMainWindow::Tf_max->Text.c_str());
    double tTf_max = atof (TMainWindow::tTf_max->Text.c_str());
    double phia_max = atof (TMainWindow::phia_max->Text.c_str());
    double tphia_max = atof (TMainWindow::tphia_max->Text.c_str());
    double phia_min = atof (TMainWindow::phia_min->Text.c_str());
```

```
double tphia_min = atof (TMainWindow::tphia_min->Text.c_str());
double vaf = atof (TMainWindow::vaf->Text.c_str());
double tVa_on = atof (TMainWindow::tVa_on->Text.c_str());
double tVa_off = atof (TMainWindow::tVa_off->Text.c_str());
double va_min1 = atof (TMainWindow::va_min1->Text.c_str());
double tVa_min1 = atof (TMainWindow::tVa_min1->Text.c_str());
double va_max1 = atof (TMainWindow::va_max1->Text.c_str());
double tVa_max1 = atof (TMainWindow::tVa_max1->Text.c_str());
double va_min2 = atof (TMainWindow::va_min2->Text.c_str());
double tVa_min2 = atof (TMainWindow::tVa_min2->Text.c_str());
double va_max2 = atof (TMainWindow::va_max2->Text.c_str());
double tVa_max2 = atof (TMainWindow::tVa_max2->Text.c_str());
double I6 = atof (TMainWindow::I6->Text.c_str());
double I7 = atof (TMainWindow::I7->Text.c_str());
double I8 = atof (TMainWindow::I8->Text.c_str());
double I9 = atof (TMainWindow::I9->Text.c_str());
double I10 = atof (TMainWindow::I10->Text.c_str());
double I11 = atof (TMainWindow::I11->Text.c_str());
double I12 = atof (TMainWindow::I12->Text.c_str());
double td;
double Ta;
double Tr;
double Tf;
double phia;
double van;
double va;
double I;
if ((0 > tTa_min) || (tTa_min >= tTa_max) || (tTa_max >= 24))
   return;
if ((0 > tTr_min) || (tTr_min >= tTr_max) || (tTr_max >= 24))
   return;
if ((0 > tTf_min) || (tTf_min >= tTf_max) || (tTf_max >= 24))
   return;
if ((0 > tphia_max) || (tphia_max >= tphia_min) || (tphia_min >= 24))
   return;
if ((0 > tVa_min1) || (tVa_min1 >= tVa_max1) || (tVa_max1 >= tVa_min2) || (
   tVa_min2 >= tVa_max2) || (tVa_max2 >= 24))
   return;
for (short i = 0; i < EnvGraph->NumPoints;) {
   td = i * 24.0 / ENVGRAPH_DATAPOINTS;
   // Calculate air temperature (Ta):
   if (td < tTa_min)
      Ta = (Ta_min + Ta_max) / 2 - (Ta_min - Ta_max) / 2 *
         cos (M_PI * (td + 24 - tTa_max) / (tTa_min + 24 - tTa_max));
   else if (td < tTa_max)
      Ta = (Ta_max + Ta_min) / 2 - (Ta_max - Ta_min) / 2 *
         cos (M_PI * (td - tTa_min) / (tTa_max - tTa_min));
   else
      Ta = (Ta_min + Ta_max) / 2 - (Ta_min - Ta_max) / 2 *
         cos (M_PI * (td - tTa_max) / (tTa_min + 24 - tTa_max));
   // Calculate temperature of surrounding radiating surfaces (Tr):
   if (td < tTr_min)
      Tr = (Tr_min + Tr_max) / 2 - (Tr_min - Tr_max) / 2 *
         cos (M_PI * (td + 24 - tTr_max) / (tTr_min + 24 - tTr_max));
   else if (td < tTr_max)
      Tr = (Tr_max + Tr_min) / 2 - (Tr_max - Tr_min) / 2 *
         cos (M_PI * (td - tTr_min) / (tTr_max - tTr_min));
   else
      Tr = (Tr_min + Tr_max) / 2 - (Tr_min - Tr_max) / 2 *
         cos (M_PI * (td - tTr_max) / (tTr_min + 24 - tTr_max));
   // Calculate temperature of the floor (Tf):
   if (td < tTf_min)
      Tf = (Tf_min + Tf_max) / 2 - (Tf_min - Tf_max) / 2 *
         cos (M_PI * (td + 24 - tTf_max) / (tTf_min + 24 - tTf_max));
   else if (td < tTf_max)
      Tf = (Tf_max + Tf_min) / 2 - (Tf_max - Tf_min) / 2 *
         cos (M_PI * (td - tTf_min) / (tTf_max - tTf_min));
   else
      Tf = (Tf_min + Tf_max) / 2 - (Tf_min - Tf_max) / 2 *
         cos (M_PI * (td - tTf_max) / (tTf_min + 24 - tTf_max));
   // Calculate relative humidity of air (phi_a):
   if (td < tphia_max)
      phia = (phia_max + phia_min) / 2 - (phia_max - phia_min) / 2 *
         cos (M_PI * (td + 24 - tphia_min) / (tphia_max + 24 - tphia_min));
   else if (td < tphia_min)
      phia = (phia_min + phia_max) / 2 - (phia_min - phia_max) / 2 *
         cos (M_PI * (td - tphia_max) / (tphia_min - tphia_max));
   else
      phia = (phia_max + phia_min) / 2 - (phia_max - phia_min) / 2 *
         cos (M_PI * (td - tphia_min) / (tphia_max + 24 - tphia_min));
   // Calculate natural air speed (van):
   if (td < tVa_min1)
      van = (va_min1 + va_max2) / 2 - (va_min1 - va_max2) / 2 *
         cos (M_PI * (td + 24 - tVa_max2) / (tVa_min1 + 24 - tVa_max2));
   else if (td < tVa_max1)
      van = (va_max1 + va_min1) / 2 - (va_max1 - va_min1) / 2 *
         cos (M_PI * (td - tVa_min1) / (tVa_max1 - tVa_min1));
```

```
        else if (td < tVa_min2)
            van = (va_min2 + va_max1) / 2 - (va_min2 - va_max1) / 2 *
                cos (M_PI * (td - tVa_max1) / (tVa_min2 - tVa_max1));
        else if (td < tVa_max2)
            van = (va_max2 + va_min2) / 2 - (va_max2 - va_min2) / 2 *
                cos (M_PI * (td - tVa_min2) / (tVa_max2 - tVa_min2));
        else
            van = (va_min1 + va_max2) / 2 - (va_min1 - va_max2) / 2 *
                cos (M_PI * (td - tVa_max2) / (tVa_min1 + 24 - tVa_max2));
        // Calculate air speed (va):
        va = van;
        if ((td > tVa_on + ROUND_OFF) && (td < tVa_off + ROUND_OFF) && (vaf > van))
            va = vaf;
        if (va < 0.15)
            va = 0.15;
        // Calculate solar radiation (I):
        double TimeFromNoon = fabs (12 - td);
        if (TimeFromNoon > 7) I = 0;
        else if (TimeFromNoon > 6) I = I6 * (7 - TimeFromNoon);
        else if (TimeFromNoon > 5) I = I6 + (I7 - I6) * (6 - TimeFromNoon);
        else if (TimeFromNoon > 4) I = I7 + (I8 - I7) * (5 - TimeFromNoon);
        else if (TimeFromNoon > 3) I = I8 + (I9 - I8) * (4 - TimeFromNoon);
        else if (TimeFromNoon > 2) I = I9 + (I10 - I9) * (3 - TimeFromNoon);
        else if (TimeFromNoon > 1) I = I10 + (I11 - I10) * (2 - TimeFromNoon);
        else I = I12 - (I12 - I11) * TimeFromNoon;
        // Update the values in the graph:
        i ++;
        EnvGraph->ThisPoint = i;
        EnvGraph->ThisSet = 1;
        EnvGraph->GraphData = Ta;
        EnvGraph->ThisSet = 2;
        EnvGraph->GraphData = Tr;
        EnvGraph->ThisSet = 3;
        EnvGraph->GraphData = Tf;
        EnvGraph->ThisSet = 4;
        EnvGraph->GraphData = phia / 2;
        EnvGraph->ThisSet = 5;
        EnvGraph->GraphData = va*10;
        EnvGraph->ThisSet = 6;
        EnvGraph->GraphData = I / 20;
    }
    EnvGraph->DrawMode = gphBlit;
}

// Validates user input for time related environment variables:
void TMainWindow::ValidateEnvTime (TEdit * TimeEdit, double Tmin, double Tmax) {
    double Time = atof (TimeEdit->Text.c_str());
    if ((Time <= Tmin) || (Time >= Tmax))
        TimeEdit->Text = (Tmin + Tmax) / 2;
}

void TEnvironment::StartSimulation (void) {
    SimulateStep (0);
}

void TEnvironment::SimulateStep (double TimeOfDay) {
    // Calculate air temperature (Ta):
    if (TimeOfDay < tTa_min)
        Ta = (Ta_min + Ta_max) / 2 - (Ta_min - Ta_max) / 2 *
            cos (M_PI * (TimeOfDay + 24 - tTa_max) / (tTa_min + 24 - tTa_max));
    else if (TimeOfDay < tTa_max)
        Ta = (Ta_max + Ta_min) / 2 - (Ta_max - Ta_min) / 2 *
            cos (M_PI * (TimeOfDay - tTa_min) / (tTa_max - tTa_min));
    else
        Ta = (Ta_min + Ta_max) / 2 - (Ta_min - Ta_max) / 2 *
            cos (M_PI * (TimeOfDay - tTa_max) / (tTa_min + 24 - tTa_max));
    // Calculate temperature of surrounding radiating surfaces (Tr):
    if (TimeOfDay < tTr_min)
        Tr = (Tr_min + Tr_max) / 2 - (Tr_max - Tr_max) / 2 *
            cos (M_PI * (TimeOfDay + 24 - tTr_max) / (tTr_min + 24 - tTr_max));
    else if (TimeOfDay < tTr_max)
        Tr = (Tr_max + Tr_min) / 2 - (Tr_max - Tr_min) / 2 *
            cos (M_PI * (TimeOfDay - tTr_min) / (tTr_max - tTr_min));
    else
        Tr = (Tr_min + Tr_max) / 2 - (Tr_min - Tr_max) / 2 *
            cos (M_PI * (TimeOfDay - tTr_max) / (tTr_min + 24 - tTr_max));
    // Calculate temperature of the floor (Tf):
    if (TimeOfDay < tTf_min)
        Tf = (Tf_min + Tf_max) / 2 - (Tf_min - Tf_max) / 2 *
            cos (M_PI * (TimeOfDay + 24 - tTf_max) / (tTf_min + 24 - tTf_max));
    else if (TimeOfDay < tTf_max)
        Tf = (Tf_max + Tf_min) / 2 - (Tf_max - Tf_min) / 2 *
            cos (M_PI * (TimeOfDay - tTf_min) / (tTf_max - tTf_min));
    else
        Tf = (Tf_min + Tf_max) / 2 - (Tf_min - Tf_max) / 2 *
            cos (M_PI * (TimeOfDay - tTf_max) / (tTf_min + 24 - tTf_max));
    // Calculate relative humidity of air (phi_a):
```

```
    if (TimeOfDay < tphia_max)
        phia = (phia_max + phia_min) / 2 - (phia_max - phia_min) / 2 *
            cos (M_PI * (TimeOfDay + 24 - tphia_min) / (tphia_max + 24 - tphia_min));
    else if (TimeOfDay < tphia_min)
        phia = (phia_min + phia_max) / 2 - (phia_min - phia_max) / 2 *
            cos (M_PI * (TimeOfDay - tphia_max) / (tphia_min - tphia_max));
    else
        phia = (phia_max + phia_min) / 2 - (phia_max - phia_min) / 2 *
            cos (M_PI * (TimeOfDay - tphia_min) / (tphia_max + 24 - tphia_min));
    // Calculate natural air speed (van):
    if (TimeOfDay < tVa_min1)
        van = (va_min1 + va_max2) / 2 - (va_min1 - va_max2) / 2 *
            cos (M_PI * (TimeOfDay + 24 - tVa_max2) / (tVa_min1 + 24 - tVa_max2));
    else if (TimeOfDay < tVa_max1)
        van = (va_max1 + va_min1) / 2 - (va_max1 - va_min1) / 2 *
            cos (M_PI * (TimeOfDay - tVa_min1) / (tVa_max1 - tVa_min1));
    else if (TimeOfDay < tVa_min2)
        van = (va_min2 + va_max1) / 2 - (va_min2 - va_max1) / 2 *
            cos (M_PI * (TimeOfDay - tVa_max1) / (tVa_min2 - tVa_max1));
    else if (TimeOfDay < tVa_max2)
        van = (va_max2 + va_min2) / 2 - (va_max2 - va_min2) / 2 *
            cos (M_PI * (TimeOfDay - tVa_min2) / (tVa_max2 - tVa_min2));
    else
        van = (va_min1 + va_max2) / 2 - (va_min1 - va_max2) / 2 *
            cos (M_PI * (TimeOfDay - tVa_max2) / (tVa_min1 + 24 - tVa_max2));
    // Calculate air speed (va):
    va = van;
    if ((TimeOfDay > tVa_on + ROUND_OFF) && (TimeOfDay < tVa_off + ROUND_OFF) && (vaf > van)) va = vaf;
    if (va < 0.15) va = 0.15;
    // Calculate solar radiation (I):
    double TimeFromNoon = fabs (12 - TimeOfDay);
    if (TimeFromNoon > 7) I = 0;
    else if (TimeFromNoon > 6) I = I6 * (7 - TimeFromNoon);
    else if (TimeFromNoon > 5) I = I6 + (I7 - I6) * (6 - TimeFromNoon);
    else if (TimeFromNoon > 4) I = I7 + (I8 - I7) * (5 - TimeFromNoon);
    else if (TimeFromNoon > 3) I = I8 + (I9 - I8) * (4 - TimeFromNoon);
    else if (TimeFromNoon > 2) I = I9 + (I10 - I9) * (3 - TimeFromNoon);
    else if (TimeFromNoon > 1) I = I10 + (I11 - I10) * (2 - TimeFromNoon);
    else I = I12 - (I12 - I11) * TimeFromNoon;
}
```

SwineSimFeedProgram.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimFeedProgramH
#define SwineSimFeedProgramH
//---------------------------------------------------------------------------

// This structure defines the variables which characterize an action in the
// daily feeding schedule:
struct TAction {
    bool Enabled;
    double Time;
    TActionType Type;
    double Amount;
};

// This structure defines the variables which characterize a feeding phase:
struct TPhase {
    char Name [NAME_SIZE];
    int DietNumber;
    bool Enabled;
    bool AdLib;
    int StartAge;
    double StartWeight;
    double InitialFeed;
    double WeeklyInc;
    TAction Action [MAX_ACTIONS];
};

// This structure defines the feed composition of a given diet:
struct TDiet {
    char Name [NAME_SIZE];
    Vector Nutrient;
    TDiet () { Nutrient = Vector (FEED_NUTRIENTS); };
};

// This class defines the feeding program:
class TFeedingProgram {
    friend TDataTransfer;
    friend void TSimulation::RecordData (TSimulatedData & SimData);
```

```
    friend void TSimulation::RecordGraphData (bool NewData);
public:
    TFeedingProgram() {
        Diet = NULL;
        NumberOfDiets = 0;
    }
    ~TFeedingProgram() {
        if (Diet) delete[] Diet;
    }
    void StartSimulation (double TimeOfDay, int Age, double Weight);
    void ContinueSimulation (double TimeOfDay, int Age, double Weight);
    void SimulateStep (double TimeOfDay, bool DayChange, int Age, double Weight);
    void RegisterFeedConsumption (double FeedIntake);
    Vector Get_Nf (void) {
        return (Diet ? Diet[Phase[ThisPhase].DietNumber].Nutrient : Vector(FEED_NUTRIENTS));
    }
    double Get_Ra (void) { return (AvailableFeed); }
    bool Get_XF (void) { return (FeedingStimulus); }
private:
    double FeedSupplied; // Rs
    double FeedConsumed; // Rc
    double FeedWasted; // Rw
    double FeedUseEfficiencyFactor; // eR
    double FeedInFeeder; // Rf
    double AvailableFeed; // Ra
    double FeedingAmount;
    bool FeedingStimulus; // XF
    int DayOfWeek;
    void ProcessAction (TAction &Action, bool AdLib);
    int ThisPhase, NextPhase, NextAction;
    int NumberOfDiets;
    bool PhaseTransitionEITHER;
    TPhase Phase [MAX_PHASES];
    void SetNumberOfDiets (int Number) {
        if (Number == NumberOfDiets) return;
        if (Diet) delete[] Diet;
        Diet = (Number > 0 ? new TDiet [Number] : NULL);
        NumberOfDiets = Number;
    }
    TDiet * Diet;
};

// This variable implements the TFeedingProgram class:
extern TFeedingProgram FeedingProgram;
//---------------------------------------------------------------------------
#endif
```

SwineSimFeedProgram.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "SwineSim.h"
#include "SwineSimFeedProgram.h"
//---------------------------------------------------------------------------

// This variable implements the TFeedingProgram class:
TFeedingProgram FeedingProgram;

// This variable defines the nutrient names used in the diet grid:
char * DietColumnTitles [FEED_NUTRIENTS + 2] = {
    "DIETS",
    "Met. Energy (MJ/kg)",
    "Protein (g/kg)",
    "Fat (g/kg)",
    "Carbohydrates (g/kg)",
    "Fiber (g/kg)",
    "Volume (cm3/kg)"
};

// Initializes other GUI parameters related to feeding phase variables:
void TMainWindow::InitFeedTab (void) {
    DietGrid->RowCount = FEED_NUTRIENTS + 2;
    for (int j = 0; j < FEED_NUTRIENTS + 2; j ++)
        DietGrid->Cells[0][j] = DietColumnTitles[j];
    DietName->Text = "";
    PhaseCurrent->Caption = 1;
    PhaseScroll->Max = MAX_PHASES;
    PhaseScroll->Position = 1;
    PhaseAction->Caption = 1;
```

```
    PhaseUpDownAction->Max = MAX_ACTIONS;
    PhaseUpDownAction->Position = 1;
    PhaseGraph->NumPoints = MAX_WEIGHT;
    PhaseGraph->NumSets = MAX_PHASES + 1;
    PhaseGraph->LabelEvery = 20;
    PhaseGraph->TickEvery = 10;
    PhaseGraph->LeftTitle = "Weight\n(kg)";
    for (short i = 1; i <= MAX_WEIGHT; i += (short) 20) {
        PhaseGraph->ThisPoint = i;
        PhaseGraph->LabelText = MAX_WEIGHT + 1 - i;
    }
    PhaseGraph->Palette = gphDefault;
    PhaseGraph->ThisSet = 1; PhaseGraph->ColorData = 14;
    PhaseGraph->ThisSet = 2; PhaseGraph->ColorData = 10;
    PhaseGraph->ThisSet = 3; PhaseGraph->ColorData = 12;
    PhaseGraph->ThisSet = 4; PhaseGraph->ColorData = 9;
    PhaseGraph->ThisSet = 5; PhaseGraph->ColorData = 6;
    PhaseGraph->ThisSet = 6; PhaseGraph->ColorData = 3;
    PhaseGraph->ThisSet = 7; PhaseGraph->ColorData = 5;
    PhaseGraph->ThisSet = 8; PhaseGraph->ColorData = 2;
    PhaseGraph->ThisSet = 9; PhaseGraph->ColorData = 4;
    PhaseGraph->ThisSet = 10; PhaseGraph->ColorData = 1;
}

// Reads all feeding program variables from the disk file to memory:
void TDataTransfer::ReadFeedingProgramFromFile (TIniFile * IniFile) {
    int i, j;
    System::AnsiString Section = "FeedingProgram";
    System::AnsiString VarName;
    FeedingProgram.FeedUseEfficiencyFactor = atof (IniFile->ReadString (Section,
        "FeedUseEfficiencyFactor", 0.95).c_str());
    int NumberOfDiets = IniFile->ReadInteger (Section, "NumberOfDiets", 0);
    FeedingProgram.SetNumberOfDiets (NumberOfDiets);
    for (i = 0; i < NumberOfDiets; i ++) {
        VarName = "Diet[";
        VarName = VarName + i + "].";
        strncpy (FeedingProgram.Diet[i].Name, IniFile->ReadString (Section,
            VarName + "Name", 0).c_str(), NAME_SIZE-1);
        FeedingProgram.Diet[i].Name[NAME_SIZE-1] = 0;
        for (j = 0; j < FEED_NUTRIENTS; j ++)
            FeedingProgram.Diet[i].Nutrient[j] = atof (IniFile->ReadString (Section,
                VarName + "Nutrient[" + j + "]", 0).c_str());
    }
    FeedingProgram.PhaseTransitionEITHER =
        IniFile->ReadBool (Section, "PhaseTransitionEITHER", false);
    for (i = 0; i < MAX_PHASES; i ++) {
        VarName = "Phase";
        strncpy (FeedingProgram.Phase[i].Name, (IniFile->ReadString (Section,
            VarName+"["+i+"].Name", VarName+" "+(i+1))).c_str(), NAME_SIZE-1);
        FeedingProgram.Phase[i].Name[NAME_SIZE-1] = 0;
        VarName = VarName + "[" + i + "].";
        FeedingProgram.Phase[i].DietNumber = IniFile->ReadInteger (Section,
            VarName + "DietNumber", -1);
        FeedingProgram.Phase[i].Enabled = IniFile->ReadBool (Section,
            VarName + "Enabled", (i ? false : true));
        FeedingProgram.Phase[i].AdLib = IniFile->ReadBool (Section, VarName + "AdLib", true);
        FeedingProgram.Phase[i].StartAge = IniFile->ReadInteger (Section,
            VarName + "StartingAge", i * 20);
        FeedingProgram.Phase[i].StartWeight = atof (IniFile->ReadString (Section,
            VarName + "StartingWeight", i * 10).c_str());
        FeedingProgram.Phase[i].InitialFeed = atof (IniFile->ReadString (Section,
            VarName + "InitialFeeding", 0).c_str());
        FeedingProgram.Phase[i].WeeklyInc = atof (IniFile->ReadString (Section,
            VarName + "WeeklyIncrement", 0).c_str());
        VarName += "Action[";
        for (j = 0; j < MAX_ACTIONS; j ++) {
            FeedingProgram.Phase[i].Action[j].Enabled = IniFile->ReadBool (Section,
                VarName + j + "].Enabled", (i || j ? false : true));
            FeedingProgram.Phase[i].Action[j].Time = atof (IniFile->ReadString (Section,
                VarName + j + "].TimeOfDay", 0).c_str());
            FeedingProgram.Phase[i].Action[j].Type = (TActionType) (IniFile->ReadInteger (Section,
                VarName + j +"].ActionType", (i || j ? -1 : actionLevel)));
            FeedingProgram.Phase[i].Action[j].Amount = atof (IniFile->ReadString (Section,
                VarName + j + "].PercentOfDailyTotal", 0).c_str());
        }
    }
}

// Writes all feeding program variables from memory to the disk file:
void TDataTransfer::WriteFeedingProgramToFile (TIniFile * IniFile) {
    int i, j;
    System::AnsiString Section = "FeedingProgram";
    System::AnsiString VarName;
    IniFile->WriteString (Section, "FeedUseEfficiencyFactor",
        FeedingProgram.FeedUseEfficiencyFactor);
    IniFile->WriteInteger (Section, "NumberOfDiets", FeedingProgram.NumberOfDiets);
    for (i = 0; i < FeedingProgram.NumberOfDiets; i ++) {
```

```
        VarName = "Diet[";
        VarName = VarName + i + "].";
        IniFile->WriteString (Section, VarName + "Name", FeedingProgram.Diet[i].Name);
        for (j = 0; j < FEED_NUTRIENTS; j ++)
            IniFile->WriteString (Section, VarName + "Nutrient[" + j + "]",
                FeedingProgram.Diet[i].Nutrient[j]);
    }
    IniFile->WriteBool (Section, "PhaseTransitionEITHER", FeedingProgram.PhaseTransitionEITHER);
    for (int i = 0; i < MAX_PHASES; i ++) {
        VarName = System::AnsiString ("Phase[") + i + "].";
        IniFile->WriteString (Section, VarName + "Name", FeedingProgram.Phase[i].Name);
        IniFile->WriteInteger (Section, VarName + "DietNumber", FeedingProgram.Phase[i].DietNumber);
        IniFile->WriteBool (Section, VarName + "Enabled", FeedingProgram.Phase[i].Enabled);
        IniFile->WriteBool (Section, VarName + "AdLib", FeedingProgram.Phase[i].AdLib);
        IniFile->WriteInteger (Section, VarName + "StartingAge", FeedingProgram.Phase[i].StartAge);
        IniFile->WriteString (Section, VarName + "StartingWeight",
            FeedingProgram.Phase[i].StartWeight);
        IniFile->WriteString (Section, VarName + "InitialFeeding",
            FeedingProgram.Phase[i].InitialFeed);
        IniFile->WriteString (Section, VarName + "WeeklyIncrement",
            FeedingProgram.Phase[i].WeeklyInc);
        VarName += "Action[";
        for (int j = 0; j < MAX_ACTIONS; j ++) {
            IniFile->WriteBool (Section, VarName + j + "].Enabled",
                FeedingProgram.Phase[i].Action[j].Enabled);
            IniFile->WriteString (Section, VarName + j + "].TimeOfDay",
                FeedingProgram.Phase[i].Action[j].Time);
            IniFile->WriteInteger (Section, VarName + j + "].ActionType",
                FeedingProgram.Phase[i].Action[j].Type);
            IniFile->WriteString (Section, VarName + j + "].PercentOfDailyTotal",
                FeedingProgram.Phase[i].Action[j].Amount);
        }
    }
}

// Reads all feeding program variables from the GUI to memory:
void TDataTransfer::ReadFeedingProgramFromScreen (void) {
    int i, j;
    FeedingProgram.FeedUseEfficiencyFactor = atof (MainWindow->FeedUseEfficiency->Text.c_str());
    int NumberOfDiets = atoi (MainWindow->DietTotal->Caption.c_str());
    FeedingProgram.SetNumberOfDiets (NumberOfDiets);
    for (i = 0; i < NumberOfDiets; i ++) {
        strncpy(FeedingProgram.Diet[i].Name,MainWindow->DietGrid->Cells[i+1][0].c_str(), NAME_SIZE-1);
        FeedingProgram.Diet[i].Name[NAME_SIZE-1] = 0;
        for (j = 0; j < FEED_NUTRIENTS; j ++)
            FeedingProgram.Diet[i].Nutrient[j] = atof (MainWindow->DietGrid->Cells[i+1][j+2].c_str());
    }
    FeedingProgram.PhaseTransitionEITHER = MainWindow->PhaseTransitionType->ItemIndex;
    memmove (FeedingProgram.Phase, MainWindow->Phase, sizeof (FeedingProgram.Phase));
}

// Writes all feeding program variables from memory to the GUI:
void TDataTransfer::WriteFeedingProgramToScreen (void) {
    int i, j;
    MainWindow->FeedUseEfficiency->Text = FeedingProgram.FeedUseEfficiencyFactor;
    MainWindow->PhaseDiet->Items->Clear ();
    MainWindow->DietGrid->ColCount =
        FeedingProgram.NumberOfDiets + 1 + (FeedingProgram.NumberOfDiets == 0);
    Matrix D (POOL_NUTRIENTS, FEED_NUTRIENTS);
    Vector E (POOL_NUTRIENTS);
    for (int i = 0; i < POOL_NUTRIENTS; i ++) {
        for (int j = 0; j < FEED_NUTRIENTS; j ++)
            D[i][j] = atof (MainWindow->DigestionGrid->Cells[j+1][i+1].c_str());
    }
    E [POOL_PROTEIN] = atof (MainWindow->EMP->Text.c_str());
    E [POOL_FAT] = atof (MainWindow->EML->Text.c_str());
    E [POOL_CARBS] = atof (MainWindow->EMG->Text.c_str());
    Vector ED = E * D;
    for (i = 0; i < FeedingProgram.NumberOfDiets; i ++) {
        MainWindow->PhaseDiet->Items->Add (FeedingProgram.Diet[i].Name);
        MainWindow->DietGrid->Cells[i+1][0] = FeedingProgram.Diet[i].Name;
        double Energy = 0;
        for (j = 0; j < FEED_NUTRIENTS; j ++) {
            MainWindow->DietGrid->Cells[i+1][j+2] = FeedingProgram.Diet[i].Nutrient[j];
            Energy += FeedingProgram.Diet[i].Nutrient[j] * ED [j];
        }
        MainWindow->DietGrid->Cells[i+1][1] =
            AnsiString::FloatToStrF (Energy / 1000, AnsiString::sffFixed, 5, 3);
    }
    if (! FeedingProgram.NumberOfDiets) {
        for (j = 0; j < FEED_NUTRIENTS + 2; j ++)
            MainWindow->DietGrid->Cells[1][j] = "";
    }
    MainWindow->DietTotal->Caption =
        (int) MainWindow->DietGrid->ColCount - 1 - (MainWindow->DietGrid->Cells[1][0] == "");
    MainWindow->DietName->Text = MainWindow->DietGrid->Cells[MainWindow->DietGrid->Col][0];
    MainWindow->PhaseTransitionType->ItemIndex = FeedingProgram.PhaseTransitionEITHER;
```

```
    memmove (MainWindow->Phase, FeedingProgram.Phase, sizeof (MainWindow->Phase));
    MainWindow->UpdatePhaseFields ();
    MainWindow->UpdatePhaseGraph ();
}

// Upgates the screen variables for the feeding phases:
void TMainWindow::UpdatePhaseFields (void) {
    int i, j;
    for (i = j = 0; i < MAX_PHASES; i ++)
      if (Phase[i].Enabled) j ++;
    PhaseTotal->Caption = j;
    i = atoi (PhaseCurrent->Caption.c_str()) - 1;
    j = atoi (PhaseAction->Caption.c_str()) - 1;
    PhaseName->Text = Phase[i].Name;
    PhaseDiet->ItemIndex = Phase[i].DietNumber;
    PhaseEnabled->Checked = Phase[i].Enabled;
    PhaseAdLib->Checked = Phase[i].AdLib;
    PhaseStartAge->Text = Phase[i].StartAge;
    PhaseStartWeight->Text = Phase[i].StartWeight;
    PhaseInitialFeed->Text = Phase[i].InitialFeed;
    PhaseWeeklyInc->Text = Phase[i].WeeklyInc;
    PhaseActionEnabled->Checked = Phase[i].Action[j].Enabled;
    PhaseTimeOfAction->Text = Phase[i].Action[j].Time;
    PhaseActionType->ItemIndex = Phase[i].Action[j].Type;
    PhaseActionAmount->Text = Phase[i].Action[j].Amount;
    i = PhaseEnabled->Checked;
    Label40->Enabled = i;
    Label41->Enabled = i;
    Label42->Enabled = i;
    Label43->Enabled = i;
    Label44->Enabled = i;
    Label45->Enabled = i;
    Label50->Enabled = i;
    PhaseAction->Enabled = i;
    PhaseName->Enabled = i;
    PhaseDiet->Enabled = i;
    PhaseAdLib->Enabled = i;
    PhaseStartAge->Enabled = i;
    PhaseStartWeight->Enabled = i;
    PhaseActionGroup->Enabled = i;
    PhaseActionEnabled->Enabled = i;
    PhaseUpDownAction->Enabled = i;
    i = PhaseEnabled->Checked && ! PhaseAdLib->Checked;
    Label46->Enabled = i;
    Label47->Enabled = i;
    Label48->Enabled = i;
    Label49->Enabled = i;
    PhaseInitialFeed->Enabled = i;
    PhaseWeeklyInc->Enabled = i;
    i = PhaseEnabled->Checked && PhaseActionEnabled->Checked;
    Label51->Enabled = i;
    Label52->Enabled = i;
    Label53->Enabled = i;
    PhaseTimeOfAction->Enabled = i;
    PhaseActionType->Enabled = i;
    i = PhaseEnabled->Checked && PhaseActionEnabled->Checked && ! PhaseAdLib->Checked &&
        ((PhaseActionType->ItemIndex == actionLevel) || (PhaseActionType->ItemIndex == actionSupply));
    Label54->Enabled = i;
    Label55->Enabled = i;
    PhaseActionAmount->Enabled = i;
}

// Upgates the graph of the feeding phases:
void TMainWindow::UpdatePhaseGraph (void) {
    for (short i = 1; i <= PhaseGraph->NumPoints; i ++) {
        PhaseGraph->ThisPoint = i;
        double weight = MAX_WEIGHT - i;
        PhaseGraph->ThisSet = MAX_PHASES + 1;
        PhaseGraph->GraphData = MAX_AGE;
        int age = MAX_AGE;
        for (short j = MAX_PHASES; j > 0;) {
            PhaseGraph->ThisSet = j;
            j --;
            if (Phase[j].Enabled && Phase[j].StartAge < MAX_AGE) {
                if (PhaseTransitionType->ItemIndex == 0) {
                    if (weight >= Phase[j].StartWeight) age = Phase[j].StartAge;
                } else {
                    age = (weight >= Phase[j].StartWeight ? 0 : Phase[j].StartAge);
                }
            }
            PhaseGraph->GraphData = age;
        }
    }
    for (short j = MAX_PHASES; j > 0;) {
        PhaseGraph->ThisSet = j;
        j --;
        PhaseGraph->LegendText = " ";
```

```
        if (Phase[j].Enabled)
            PhaseGraph->LegendText = Phase[j].Name;
    }
    PhaseGraph->DrawMode = gphBlit;
}

// Validates a change in starting age of the phase:
void TMainWindow::ValidatePhaseAge (int CurPhase) {
    int i;
    Phase[CurPhase].StartAge = atoi (PhaseStartAge->Text.c_str());
    if (Phase[CurPhase].StartAge < 0)
        Phase[CurPhase].StartAge = 0;
    for (i = 0; i < CurPhase; i ++)
        if (Phase[i].StartAge > Phase[CurPhase].StartAge)
            Phase[i].StartAge = Phase[CurPhase].StartAge;
    for (i = CurPhase + 1; i < MAX_PHASES; i ++)
        if (Phase[i].StartAge < Phase[CurPhase].StartAge)
            Phase[i].StartAge = Phase[CurPhase].StartAge;
    PhaseStartAge->Text = Phase[CurPhase].StartAge;
}

// Validates a change in starting weight of the phase:
void TMainWindow::ValidatePhaseWeight (int CurPhase) {
    int i;
    Phase[CurPhase].StartWeight = atof (PhaseStartWeight->Text.c_str());
    if (Phase[CurPhase].StartWeight < 0)
        Phase[CurPhase].StartWeight = 0;
    for (i = 0; i < CurPhase; i ++)
        if (Phase[i].StartWeight > Phase[CurPhase].StartWeight)
            Phase[i].StartWeight = Phase[CurPhase].StartWeight;
    for (i = CurPhase + 1; i < MAX_PHASES; i ++)
        if (Phase[i].StartWeight < Phase[CurPhase].StartWeight)
            Phase[i].StartWeight = Phase[CurPhase].StartWeight;
    PhaseStartWeight->Text = Phase[CurPhase].StartWeight;
}

// Validates a change in the initial feeding amount:
void TMainWindow::ValidatePhaseInitialFeed (int CurPhase) {
    Phase[CurPhase].InitialFeed = atof (PhaseInitialFeed->Text.c_str());
    if (Phase[CurPhase].InitialFeed < 0)
        Phase[CurPhase].InitialFeed = 0;
    PhaseInitialFeed->Text = Phase[CurPhase].InitialFeed;
}

// Validates a change in time of day of the action:
void TMainWindow::ValidatePhaseTimeOfAction (int CurPhase, int CurAction) {
    int i;
    Phase[CurPhase].Action[CurAction].Time =
        atof (PhaseTimeOfAction->Text.c_str());
    if (Phase[CurPhase].Action[CurAction].Time < 0)
        Phase[CurPhase].Action[CurAction].Time =
            (CurAction ? Phase[CurPhase].Action[CurAction-1].Time : 0);
    if (Phase[CurPhase].Action[CurAction].Time >= 24)
        Phase[CurPhase].Action[CurAction].Time =
            (CurAction < MAX_ACTIONS - 1 ? Phase[CurPhase].Action[CurAction+1].Time :
            (CurAction ? Phase[CurPhase].Action[CurAction-1].Time : 0));
    for (i = 0; i < CurAction; i ++)
        if (Phase[CurPhase].Action[i].Time>Phase[CurPhase].Action[CurAction].Time)
            Phase[CurPhase].Action[i].Time=Phase[CurPhase].Action[CurAction].Time;
    for (i = CurAction + 1; i < MAX_ACTIONS; i ++)
        if (Phase[CurPhase].Action[i].Time<Phase[CurPhase].Action[CurAction].Time)
            Phase[CurPhase].Action[i].Time=Phase[CurPhase].Action[CurAction].Time;
    PhaseTimeOfAction->Text = Phase[CurPhase].Action[CurAction].Time;
}

// Validates a change in the action amount as a percent of daily total:
void TMainWindow::ValidatePhaseActionAmount (int CurPhase, int CurAction) {
    Phase[CurPhase].Action[CurAction].Amount = atof (PhaseActionAmount->Text.c_str());
    if (Phase[CurPhase].Action[CurAction].Amount < 0)
        Phase[CurPhase].Action[CurAction].Amount = 0;
    PhaseActionAmount->Text = Phase[CurPhase].Action[CurAction].Amount;
}

void TMainWindow::DietAdd (void) {
    int i;
    if (DietName->Text==""||DietName->Text==DietGrid->Cells[DietGrid->Col][0]) {
        DietName->SetFocus();
        return;
    }
    for (i = 1; i < DietGrid->ColCount; i ++)
        if (DietGrid->Cells[i][0] == DietName->Text) {
            DietGrid->Col = i;
            DietGrid->SetFocus();
            return;
        }
    PhaseDiet->Items->Add (DietName->Text);
    if (DietGrid->Cells[1][0] != "")
```

```
        DietGrid->ColCount ++;
    DietGrid->Cells[DietGrid->ColCount-1][0] = DietName->Text;
    DietGrid->Cells[DietGrid->ColCount-1][1] = 0;
    DietGrid->Col = DietGrid->ColCount - 1;
    DietTotal->Caption = (int) DietGrid->ColCount-1-(DietGrid->Cells[1][0]=="");
    DietGrid->SetFocus();
}

void TMainWindow::DietInsert (void) {
    int i, j;
    if (DietName->Text==""||DietName->Text==DietGrid->Cells[DietGrid->Col][0]) {
        DietName->SetFocus();
        return;
    }
    for (i = 1; i < DietGrid->ColCount; i ++)
        if (DietGrid->Cells[i][0] == DietName->Text) {
            DietGrid->Col = i;
            DietGrid->SetFocus();
            return;
        }
    PhaseDiet->Items->Insert (DietGrid->Col-1, DietName->Text);
    for (i = 0; i < MAX_PHASES; i ++)
        if (Phase[i].DietNumber >= DietGrid->Col-1)
            Phase[i].DietNumber ++;
    if (DietGrid->Cells[1][0] != "") {
        DietGrid->ColCount ++;
        for (i = DietGrid->ColCount - 1; i > DietGrid->Col; i --)
            for (j = 0; j < DietGrid->RowCount; j ++)
                DietGrid->Cells[i][j] = DietGrid->Cells[i-1][j];
        for (j = 1; j < DietGrid->RowCount; j ++)
            DietGrid->Cells[DietGrid->Col][j] = "";
    }
    DietGrid->Cells[DietGrid->Col][0] = DietName->Text;
    DietGrid->Cells[DietGrid->Col][1] = 0;
    DietTotal->Caption = (int) DietGrid->ColCount - 1 - (DietGrid->Cells[1][0] == "");
    DietGrid->SetFocus();
}

void TMainWindow::DietDelete (void) {
    int i, j;
    if (DietGrid->Cells[1][0] != "") {
        PhaseDiet->Items->Delete (DietGrid->Col-1);
        for (i = 0; i < MAX_PHASES; i ++) {
            if (Phase[i].DietNumber == DietGrid->Col-1)
                Phase[i].DietNumber = -1;
            if (Phase[i].DietNumber > DietGrid->Col-1)
                Phase[i].DietNumber --;
        }
    }
    for (i = DietGrid->Col; i < DietGrid->ColCount-1; i ++)
        for (j = 0; j < DietGrid->RowCount; j ++)
            DietGrid->Cells[i][j] = DietGrid->Cells[i+1][j];
    for (j = 0; j < DietGrid->RowCount; j ++)
        DietGrid->Cells[DietGrid->ColCount-1][j] = "";
    if (DietGrid->ColCount > 2)
        DietGrid->ColCount --;
    DietName->Text = DietGrid->Cells[DietGrid->Col][0];
    DietTotal->Caption = (int) DietGrid->ColCount - 1 - (DietGrid->Cells[1][0] == "");
    DietGrid->SetFocus();
}

void TMainWindow::DietRename (void) {
    int i;
    if (DietName->Text == "" || DietName->Text == DietGrid->Cells[DietGrid->Col][0]) {
        DietName->SetFocus();
        return;
    }
    for (i = 1; i < DietGrid->ColCount; i ++)
        if (DietGrid->Cells[i][0] == DietName->Text) {
            DietGrid->Col = i;
            DietGrid->SetFocus();
            return;
        }
    i = PhaseDiet->ItemIndex;
    PhaseDiet->Items->Strings[DietGrid->Col-1] = DietName->Text;
    PhaseDiet->ItemIndex = i;
    DietGrid->Cells[DietGrid->Col][0] = DietName->Text;
    DietTotal->Caption = (int) DietGrid->ColCount - 1 - (DietGrid->Cells[1][0] == "");
    DietGrid->SetFocus();
}

void TFeedingProgram::StartSimulation (double TimeOfDay, int Age, double Weight) {
    FeedSupplied = FeedConsumed = FeedWasted = 0;
    FeedInFeeder = AvailableFeed = 0;
    FeedingStimulus = false;
    ThisPhase = -1;
    ContinueSimulation (TimeOfDay, Age, Weight);
```

```
    if (Phase[ThisPhase].AdLib)
        AvailableFeed = LOTS_OF_FEED;
}

void TFeedingProgram::ContinueSimulation (double TimeOfDay, int Age, double Weight) {
    // Remember in which phase the animal was in:
    int LastThisPhase = ThisPhase;
    // Test the input data for inconsistencies:
    for (ThisPhase = 0; ThisPhase < MAX_PHASES; ThisPhase ++) {
        if (Phase[ThisPhase].Enabled) {
            if ((Phase[ThisPhase].DietNumber < 0) || (Phase[ThisPhase].DietNumber >= NumberOfDiets))
                throw ("No diet is specified in one of the enabled phases!");
            if (!( Phase[ThisPhase].AdLib) && (Phase[ThisPhase].InitialFeed <= 0))
                throw ("Zero feeding in one of the enabled phases!");
            double TotalFeedSupplied = 0;
            for (NextAction = 0; NextAction < MAX_ACTIONS; NextAction ++) {
                if (Phase[ThisPhase].Action[NextAction].Enabled) {
                    if ((Phase[ThisPhase].Action[NextAction].Type == actionLevel) ||
                        (Phase[ThisPhase].Action[NextAction].Type == actionSupply)) {
                        TotalFeedSupplied += (Phase[ThisPhase].AdLib ? 1 :
                            Phase[ThisPhase].Action[NextAction].Amount);
                    }
                }
            }
            if (TotalFeedSupplied <= 0)
                throw ("No Level feed or Supply feed action in one of the enabled phases!");
        }
    }
    // Find the first feeding phase that is enabled:
    for (ThisPhase = 0; ThisPhase < MAX_PHASES; ThisPhase ++) {
        if (Phase[ThisPhase].Enabled)
            break;
    }
    // If there are no phases enabled, throw an exception:
    if (ThisPhase == MAX_PHASES)
        throw ("No feeding phases are enabled!");
    // If the age and weight of the animal are too small to be in the first
    // enabled feeding phase, throw an exception:
    if (PhaseTransitionEITHER ?
        ((Age < Phase[ThisPhase].StartAge) && (Weight < Phase[ThisPhase].StartWeight)) :
        ((Age < Phase[ThisPhase].StartAge) || (Weight < Phase[ThisPhase].StartWeight))) {
        throw ("The animal is not big or old enough to be in the first feeding phase!");
    }
    // Find the next phase that is enabled, or determine that there are
    // no more enabled phases left:
    for (NextPhase = ThisPhase + 1; NextPhase < MAX_PHASES; NextPhase ++) {
        if (Phase[NextPhase].Enabled) {
            // If the phase is enabled, check to make sure the animal should not
            // already be in it, and if so, update the pointer to the current
            // phase; otherwise consider the next phase to be found:
            if (PhaseTransitionEITHER ?
                ((Age >= Phase[NextPhase].StartAge) || (Weight >= Phase[NextPhase].StartWeight)) :
                ((Age >= Phase[NextPhase].StartAge) && (Weight >= Phase[NextPhase].StartWeight))) {
                ThisPhase = NextPhase;
            } else {
                break;
            }
        }
    }
    // Skip all the actions that occured before the current time of day:
    for (NextAction = 0; NextAction < MAX_ACTIONS; NextAction ++)
        if (TimeOfDay - ROUND_OFF < Phase[ThisPhase].Action[NextAction].Time)
            break;
    // Initialize other variables:
    if (ThisPhase != LastThisPhase) {
        FeedingAmount = Phase[ThisPhase].InitialFeed;
        DayOfWeek = 0;
    }
}

void TFeedingProgram::ProcessAction (TAction &Action, bool AdLib) {
    // Process only the actions that are enabled:
    if (Action.Enabled) {
        switch (Action.Type) {
            case actionRemove:
                AvailableFeed = 0;
            break;
            case actionRestore:
                if (AdLib)
                    AvailableFeed = LOTS_OF_FEED;
                else
                    AvailableFeed = FeedInFeeder * FeedUseEfficiencyFactor;
                FeedingStimulus = true;
            break;
            case actionLevel:
                if (AdLib) {
                    AvailableFeed = LOTS_OF_FEED;
```

```
                FeedInFeeder = 0;
            } else {
                double Amount = Action.Amount * FeedingAmount / 100;
                if (Amount > FeedInFeeder)
                    FeedSupplied += Amount - FeedInFeeder;
                else
                    FeedWasted += FeedInFeeder - Amount;
                FeedInFeeder = Amount;
                AvailableFeed = FeedInFeeder * FeedUseEfficiencyFactor;
            }
            FeedingStimulus = true;
            break;
            case actionSupply:
                if (AdLib) {
                    AvailableFeed = LOTS_OF_FERD;
                    FeedInFeeder = 0;
                } else {
                    double Amount = Action.Amount * FeedingAmount / 100;
                    FeedSupplied += Amount;
                    FeedInFeeder += Amount;
                    AvailableFeed = FeedInFeeder * FeedUseEfficiencyFactor;
                }
                FeedingStimulus = true;
            break;
            case actionDiscard:
                if (! AdLib)
                    FeedWasted += FeedInFeeder;
                FeedInFeeder = 0;
                AvailableFeed = 0;
            break;
        }
    }
}

void TFeedingProgram::SimulateStep (double TimeOfDay, bool DayChange, int Age, double Weight) {
    // There will be no feeding stimulus unless feed was restored, supplied or leveled:
    FeedingStimulus = false;
    // If the day changed, some special processing is needed:
    if (DayChange) {
        // First execute all the actions at the end of the previous day:
        while (NextAction < MAX_ACTIONS) {
            ProcessAction (Phase[ThisPhase].Action[NextAction], Phase[ThisPhase].AdLib);
            NextAction ++;
        }
        // Update day of the week, incrementing feed amount if the week changed:
        DayOfWeek ++;
        if (DayOfWeek == 7) {
            DayOfWeek = 0;
            FeedingAmount += Phase[ThisPhase].WeeklyInc;
        }
        // If there are still phases left, test age and weight for a phase
        // transition; this transition may be when the first condition (age or
        // weight) is satisfied, or only after both conditions are satisfied,
        // depending on the user's choice:
        if (NextPhase < MAX_PHASES) {
            if (PhaseTransitionEITHER ?
                ((Age >= Phase[NextPhase].StartAge) || (Weight >= Phase[NextPhase].StartWeight)) :
                ((Age >= Phase[NextPhase].StartAge) && (Weight >= Phase[NextPhase].StartWeight))) {
                ThisPhase = NextPhase;
                // Find the next phase that is enabled, or determine that there are
                // no more enabled phases left:
                for (NextPhase ++; NextPhase < MAX_PHASES; NextPhase ++) {
                    if (Phase[NextPhase].Enabled) {
                        // If the phase is enabled, check to make sure the animal should not
                        // already be in it, and if so, update the pointer to the current
                        // phase; otherwise consider the next phase to be found:
                        if (PhaseTransitionEITHER ?
                            ((Age >= Phase[NextPhase].StartAge) ||
                            (Weight >= Phase[NextPhase].StartWeight)) :
                            ((Age >= Phase[NextPhase].StartAge) &&
                            (Weight >= Phase[NextPhase].StartWeight))) {
                            ThisPhase = NextPhase;
                        } else {
                            break;
                        }
                    }
                }
                FeedingAmount = Phase[ThisPhase].InitialFeed;
                DayOfWeek = 0;
            }
        }
        // Reset the action pointer:
        NextAction = 0;
    }
    // Execute all the actions that should have occured during the time step:
    while ((NextAction<MAX_ACTIONS)&&(TimeOfDay-ROUND_OFF>Phase[ThisPhase].Action[NextAction].Time)){
        ProcessAction(Phase[ThisPhase].Action[NextAction], Phase[ThisPhase].AdLib);
```

```
        NextAction ++;
    }
}

void TFeedingProgram::RegisterFeedConsumption (double FeedIntake) {
    // Register the amount of feed consumed and wasted:
    FeedConsumed += FeedIntake;
    FeedWasted += FeedIntake * (1 / FeedUseEfficiencyFactor - 1);
    // If feeding is Ad Lib, update the amount of feed in the feeder and
    // available feed, otherwise, the feed in the feeder will remain zero and
    // the available feed will remain a large number; if feeding is Ad Lib, the
    // amount of feed supplied is calculated on each time step, as the sum of
    // feed consumed and feed wasted:
    if (! Phase[ThisPhase].AdLib) {
        FeedInFeeder -= FeedIntake / FeedUseEfficiencyFactor;
        AvailableFeed -= FeedIntake;
    } else {
        FeedSupplied += FeedIntake / FeedUseEfficiencyFactor;
    }
}
```

SwineSimHeatBalance.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimHeatBalanceH
#define SwineSimHeatBalanceH

#include <math.h>

class THeatBalance {
    friend TDataTransfer;
    friend void TSimulation::RecordData (TSimulatedData & SimData);
public:
    void StartSimulation (double TimeStep);
    void SimulateStep (double TimeStep);
    double TNlevel (double X0, double X1) {
        // used when a parameter changes under thermoneutral conditions
        if (Tb <= Tc) return (X0);
        if (Tb >= Te) return (X1);
        return ((X0 * (Te - Tb) + X1 * (Tb - Tc)) / (Te - Tc));
    };
    double HSlevel (double Y0, double Y1) {
        // used when the parameter changes under heat stress
        if (Tb <= Te) return (Y0);
        if (Tb >= Th) return (Y1);
        return ((Y0 * (Th - Tb) + Y1 * (Tb - Te)) / (Th - Te));
    };
    double HSrise (double Z0, double zr) {
        // used when the parameter increases indefinitely under heat stress
        if (Tb <= Te) return (Z0);
        return (Z0 + zr * (Tb - Te));
    };
    double CStemp(void) {
        // returns the body temperature deficit under cold stress
        return (Tb < Tc ? Tc - Tb : 0);
    };
    static const double Cf0_List [5];
    static const double cpw;
    THeatBalance () { Tb = 39; };
private:
    // State variables (must be initialized):
    double Tb;
    double Tsf;
    double Tse;
    double Tss;
    double kf;
    // Variables needed for the graphs (otherwise could be local):
    double qRf, qRe, qRs, qv, qL, ke, ks, Ab;
    double qRe_rad, qRe_conv, qRe_evap;
    double qRs_rad, qRs_conv, qRs_evap, qRs_sol;
    double QL;
    double t_cold, t_hot, t_evap, t_neutral;
    double TT_cold, TT_hot, TT_evap, TT_neutral;
    // Heat Balance parameters:
    double Th;
    double Te;
    double Tc;
    double Ct0_;
    double Ct1_;
    bool FloorTypeEnabled;
    int FloorType;
```

```
    double Cf0;
    int N;
    static const double StefanBoltzmann;
    double el;
    double er;
    static const double hL;
    double rwp0;
    double rwpl;
    double rvw0;
    double rvwl;
    double theta;
    double es;
    double kA;
    double ka0;
    double kal;
    double kf0;
    double kfl;
    double ks0;
    double ksl;
    double ksa;
    double kx;
    double rv0_;
    double rvl_;
    double Rw0;
    double Rwl;
    double cpl;
    // Convective heat and vapor transfer coefficients:
    double ConvectiveHeatTransferCoefficient (double va, double Wb) {
       return (15.7 * pow (va, 0.6) / pow (Wb, 0.13));
    }
    double ConvectiveVaporTransferCoefficient (double va, double Wb) {
       return (9.85e-5 * pow (va, 0.6) / pow (Wb, 0.13));
    }
};

// This variable implements the THeatBalance class:
extern THeatBalance HeatBalance;
//---------------------------------------------------------------------------
#endif
```

SwineHeatBalance.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include <math.h>
#include "SwineSim.h"
#include "SwineSimHeatBalance.h"
//---------------------------------------------------------------------------

// This variable implements the THeatBalance class:
THeatBalance HeatBalance;

// The list of pre-defined thermal conductances of the floor:
const double THeatBalance::Cf0_List [5] = { 10.16, 5.93, 3.09, 1.78, 1.42 };
// The Stefan-Boltzmann constant (W/m2.K4):
const double THeatBalance::StefanBoltzmann = 5.6697e-8;
// The latent heat of evaporation of water (J/g):
const double THeatBalance::hL = 2425;
// The specific heat of water (kJ/kg.C):
const double THeatBalance::cpw = 4.1819;

// Function that initializes GUI parameters related to heat balance variables;
// should be called once when the program starts:
void TMainWindow::InitHeatTab (void) {
   DataTransfer.SetHeatBalanceDescription ();
}

// Macro that sets the description of one variable that is transfered between
// disk file, memory and GUI:
#define SHD(i,ADDRESS,EDIT,NAME,DEFAULT) {          \
   HeatBalanceDescription[i].Address = (ADDRESS); \
   HeatBalanceDescription[i].Edit = (EDIT);       \
   HeatBalanceDescription[i].Name = (NAME);       \
   HeatBalanceDescription[i].Default = (DEFAULT); \
}

// Variable that implements the description of all variables that are transfered
// between disk file, memory and GUI:
TVariableDescription TDataTransfer::HeatBalanceDescription [29];
```

```
// Function that sets the description of all variables that are transfered
// between disk file, memory and GUI:
void TDataTransfer::SetHeatBalanceDescription (void) {
   SHD (0, &(HeatBalance.Th), MainWindow->Th, "UpperCriticalTemperature", 40.5);
   SHD (1, &(HeatBalance.Te), MainWindow->Te, "EvaporativeCriticalTemperature", 39.5);
   SHD (2, &(HeatBalance.Tc), MainWindow->Tc, "LowerCriticalTemperature", 38.5);
   SHD (3, &(HeatBalance.Ct0_), MainWindow->Ct0_, "MinimumConductanceOfSkin", 50);
   SHD (4, &(HeatBalance.Ct1_), MainWindow->Ct1_, "MaximumConductanceOfSkin", 200);
   SHD (5, &(HeatBalance.Cf0), MainWindow->Cf0, "ThermalConductanceOfFloor",HeatBalance.Cf0_List[0]);
   SHD (6, &(HeatBalance.el), MainWindow->el, "LongWaveEmissivityOfSkin", 0.95);
   SHD (7, &(HeatBalance.er), MainWindow->er, "LongWaveEmissivityOfWalls", 0.95);
   SHD (8, &(HeatBalance.rwp0), MainWindow->rwp0, "MinimumSkinWettedByPerspiration", 0.037);
   SHD (9, &(HeatBalance.rwp1), MainWindow->rwp1, "MaximumSkinWettedByPerspiration", 0.042);
   SHD (10, &(HeatBalance.rww0), MainWindow->rww0, "MinimumSkinWettedByWallowing", 0);
   SHD (11, &(HeatBalance.rww1), MainWindow->rww1, "MaximumSkinWettedByWallowing", 0.15);
   SHD (12, &(HeatBalance.theta), MainWindow->theta, "AngleBetweenSkinAndSun", 40);
   SHD (13, &(HeatBalance.es), MainWindow->es, "ShortWaveEmissivityOfSkin", 0.50);
   SHD (14, &(HeatBalance.ka), MainWindow->ka, "CoefficientOfBodySurface", 0.09);
   SHD (15, &(HeatBalance.ka0), MainWindow->ka0, "SkinInSelfContact", 0);
   SHD (16, &(HeatBalance.ka1), MainWindow->ka1, "SkinTouchingOneOtherAnimal", 0.075);
   SHD (17, &(HeatBalance.kf0), MainWindow->kf0, "MinimumSkinTouchingFloor", 0.1);
   SHD (18, &(HeatBalance.kf1), MainWindow->kf1, "MaximumSkinTouchingFloor", 0.2);
   SHD (19, &(HeatBalance.ks0), MainWindow->ks0, "MinimumSkinExposedToSunlight", 0);
   SHD (20, &(HeatBalance.ks1), MainWindow->ks1, "MaximumSkinExposedToSunlight", 0);
   SHD (21, &(HeatBalance.ksa), MainWindow->ksa, "AvailabilityOfShade", 1);
   SHD (22, &(HeatBalance.kx), MainWindow->kx, "CoefficientOfCooling", 0.6);
   SHD (23, &(HeatBalance.rv0_), MainWindow->rv0_, "MinimumVentilationRate", 0.117);
   SHD (24, &(HeatBalance.rv1_), MainWindow->rv1_, "MaximumVentilationRate", 0.583);
   SHD (25, &(HeatBalance.Rw0), MainWindow->Rw0, "MinimumWaterIngestion", 0.007);
   SHD (26, &(HeatBalance.Rw1), MainWindow->Rw1, "MaximumWaterIngestion", 0.015);
   SHD (27, &(HeatBalance.cpi), MainWindow->cpi, "FeedSpecificHeat", 2);
   SHD (28, NULL, NULL, NULL, 0);
}

// Function that reads all heat balance variables from the disk file to memory:
void TDataTransfer::ReadHeatBalanceFromFile (TIniFile * IniFile) {
   TVariableDescription * VarPtr;
   for (VarPtr = HeatBalanceDescription; VarPtr->Address; VarPtr ++)
      *(VarPtr->Address) = atof (IniFile->ReadString ("HeatBalance",
         VarPtr->Name, VarPtr->Default).c_str());
   HeatBalance.FloorTypeEnabled = IniFile->ReadBool ("HeatBalance", "FloorTypeEnabled", true);
   HeatBalance.FloorType = IniFile->ReadInteger ("HeatBalance", "FloorType", 0);
   if (HeatBalance.FloorTypeEnabled)
      HeatBalance.Cf0 = HeatBalance.Cf0_List[HeatBalance.FloorType];
   HeatBalance.N = IniFile->ReadInteger ("HeatBalance", "AnimalsPerPen", 1);
}

// Function that writes all heat balance variables from memory to the disk file:
void TDataTransfer::WriteHeatBalanceToFile (TIniFile * IniFile) {
   TVariableDescription * VarPtr;
   for (VarPtr = HeatBalanceDescription; VarPtr->Address; VarPtr ++)
      IniFile->WriteString ("HeatBalance", VarPtr->Name, *(VarPtr->Address));
   IniFile->WriteBool ("HeatBalance", "FloorTypeEnabled", HeatBalance.FloorTypeEnabled);
   IniFile->WriteInteger ("HeatBalance", "FloorType", HeatBalance.FloorType);
   IniFile->WriteInteger ("HeatBalance", "AnimalsPerPen", HeatBalance.N);
}

// Function that reads all heat balance variables from the GUI to memory:
void TDataTransfer::ReadHeatBalanceFromScreen (void) {
   TVariableDescription * VarPtr;
   for (VarPtr = HeatBalanceDescription; VarPtr->Address; VarPtr ++)
      *(VarPtr->Address) = atof (VarPtr->Edit->Text.c_str());
   HeatBalance.FloorTypeEnabled = MainWindow->BuildRbFloorType->Checked;
   HeatBalance.FloorType = MainWindow->BuildFloorType->ItemIndex;
   HeatBalance.N = atoi (MainWindow->N->Text.c_str());
}

// Function that writes all heat balance variables from memory to the GUI:
void TDataTransfer::WriteHeatBalanceToScreen (void) {
   TVariableDescription * VarPtr;
   for (VarPtr = HeatBalanceDescription; VarPtr->Address; VarPtr ++)
      VarPtr->Edit->Text = *(VarPtr->Address);
   MainWindow->BuildRbFloorType->Checked = HeatBalance.FloorTypeEnabled;
   MainWindow->BuildRbFloorConductance->Checked = ! HeatBalance.FloorTypeEnabled;
   MainWindow->BuildFloorType->ItemIndex = HeatBalance.FloorType;
   MainWindow->N->Text = HeatBalance.N;
}

// Initialize the simulation
void THeatBalance::StartSimulation (double TimeStep) {
   Tb = Tsf = Tse = Tss = (Tc + Te) / 2;
   kf = (kf0 + kf1) / 2;
   double OldTb = 0;
   for (int i = 0; (fabs (Tb - OldTb) > ROUND_OFF) && (i < 1 / TimeStep); i ++) {
      OldTb = Tb;
      SimulateStep (TimeStep);
```

```
    }
    T_cold = T_hot = T_evap = T_neutral = 0;
    TT_cold = TT_hot = TT_evap = TT_neutral = 0;
}

void THeatBalance::SimulateStep (double TimeStep) {
    double Ta = Environment.Get_Ta();
    double Tr = Environment.Get_Tr();
    double Tf = Environment.Get_Tf();
    double phia = Environment.Get_phia();
    double va = Environment.Get_va();
    double I = Environment.Get_I();
    double Wb = Metabolism.Get_Wb();
    double Ct = TNlevel (Ct0_, Ctl_) / pow (Wb, 1.0 / 3);
    double Cf = Cf0 / (pow (Wb, 1.0 / 3) * kf * pow (N, 0.5));
    Taf = (Ct * Tb + Cf * Tf) / (Ct + Cf);
    qRf = Ct * Cf * (Tb - Tf) / (Ct + Cf);
    double rwp = HSlevel (rwp0, rwp1);
    double rww = TNlevel (rww0, rww1);
    double rw = 1 - (1 - rwp) * (1 - rww);
    double pwa = phia * Environment.SaturationVaporPressure (Ta);
    double hC = ConvectiveHeatTransferCoefficient (va, Wb);
    double hE = ConvectiveVaporTransferCoefficient (va, Wb);
    double pwse = Environment.SaturationVaporPressure (Tse);
    Tse = (Tse + (Ct * Tb + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) * Tr
        + hC * Ta - rw * hL * hE * (pwse - pwa))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC)) / 2;
    pwse = Environment.SaturationVaporPressure (Tse);
    qRe_rad = Ct * (4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) * (Tb - Tr))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRe_conv = Ct * (hC * (Tb - Ta))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRe_evap = Ct * (rw * hL * hE * (pwse - pwa))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRe = qRe_rad + qRe_conv + qRe_evap;
    double pwss = Environment.SaturationVaporPressure (Tss);
    Tss = (Tss + (Ct * Tb + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) * Tr
        + hC * Ta - rw * hL * hE * (pwss - pwa) + I * sin (theta * M_PI / 180) * es)
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC)) / 2;
    pwss = Environment.SaturationVaporPressure (Tss);
    qRs_rad = Ct * (4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) * (Tb - Tr))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRs_conv = Ct * (hC * (Tb - Ta))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRs_evap = Ct * (rw * hL * hE * (pwss - pwa))
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRs_sol = Ct * (- I * sin (theta * M_PI / 180) * es)
        / (Ct + 4 * StefanBoltzmann * e1 * er * pow (Ta + 273.15, 3) + hC);
    qRs = qRs_rad + qRs_conv + qRs_evap + qRs_sol;
    Ab = kA * pow (Wb, 2.0 / 3);
    double ka = TNlevel (ka0 + (ka1 * 2 * (N - 1)) / N, 0);
    kf = (qRf > qRe ? TNlevel (kf0, kf1) : TNlevel (kfl, kf0));
    ks = TNlevel (ks1, ks0 * (1 - ksa));
    ke = 1 - (ka + kf + ks);
    if (ke < 0) { kf /= (1 - ke); ks /= (1 - ke); ke = 0; }
    double Tx = Tb + kx * (Ta - Tb);
    double hx = Environment.EnthalpyOfAir (Tx, 1);
    double ha = Environment.EnthalpyOfAir (Ta, phia);
    double rv = HSlevel (rv0_, rv1_) * Ab;
    double rhob = Environment.DensityOfAir (Tb, 1);
    qv = rv * rhob * (hx - ha);
    double Ri = Metabolism.Get_Ri();
    double Rw = HSlevel (Rw0, Rw1);
    double Ti = Environment.Get_Ti();
    double Tw = Environment.Get_Tw();
    qi = (Ri * cpi * (Tb - Ti) / TimeStep + Rw * pow (Wb, 0.75) * cpw * (Tb - Tw)) / 3.6;
    QL = ((qRf * kf + qRe * ke + qRs * ks) * Ab + qv + qi) * TimeStep * 3.6;
    double QP = Metabolism.Get_QP();
    double cpb = Metabolism.Get_cpb();
    Tb += (QP - QL) / (cpb * Wb);
    if (Tb < Tc - ROUND_OFF) {
        T_cold += TimeStep;
        TT_cold += TimeStep * Tb;
    } else if (Tb > Th + ROUND_OFF) {
        T_hot += TimeStep;
        TT_hot += TimeStep * Tb;
    } else if (Tb > Te + ROUND_OFF) {
        T_evap += TimeStep;
        TT_evap += TimeStep * Tb;
    } else {
        T_neutral += TimeStep;
        TT_neutral += TimeStep * Tb;
    }
}
```

SwineSimMetabolism.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimMetabolismH
#define SwineSimMetabolismH

//---------------------------------------------------------------------------
class TMetabolism {
    friend TDataTransfer;
    friend void TSimulation::RecordData (TSimulatedData & SimData);
    friend void TSimulation::RecordGraphData (bool NewData);
public:
    TMetabolism () {
        Np = Vector (POOL_NUTRIENTS);
        Ni = Vector (FEED_NUTRIENTS);
        EMN = Vector (POOL_NUTRIENTS);
        ENN = Vector (POOL_NUTRIENTS);
        Nlt = Vector (POOL_NUTRIENTS);
        Nft = Vector (POOL_NUTRIENTS);
        Nmr = Vector (POOL_NUTRIENTS);
        Nlc = Vector (POOL_NUTRIENTS);
        Nfc = Vector (POOL_NUTRIENTS);
        Nld = Vector (POOL_NUTRIENTS);
        DN = Matrix (POOL_NUTRIENTS, FEED_NUTRIENTS);
        DE = Vector (FEED_NUTRIENTS);
        DQ = Vector (FEED_NUTRIENTS);
    };
    double Get_Wb (void) { return (Wb); };
    double Get_QP (void) { return (QP); };
    double Get_Ri (void) { return (Ri); };
    double Get_cpb (void) { return (cpb); };
    void StartSimulation (double Weight, double Age, double TimeStep);
    void SimulateStep (double TimeStep, int Age, bool ConstantWeight);
    double LeanGrowthCurve (int Age, double LeanMass, double FatMass);
        // maximum rate of lean tissue deposition (kg/h)
    double FatGrowthCurve (int Age, double LeanMass, double FatMass);
        // maximum rate of excess fat tissue deposition (kg/h)
private:
    // State variables (need to be initialized):
    double Wlt;
    double Wft;
    Vector Np;
    bool AnimalIsEating;
    Vector Ni;
    double QP; // heat production in the last time step (kJ)
    double Ri; // feed intake in the last time step (kg)
    // Simulation parameters:
    double InitialFatLeanRatio;
    double InitialHeatProduction;
    // Animal Metabolism parameters:
    double kFL, kWT, kEB, qmax;
    double kF;
    Vector EMN;
    Vector ENN;
    Vector Nlt;
    Vector Nft;
    double Emin_;
    double Emax_;
    double kv;
    double ri_;
    double rd;
    double Rmr_;
    double kr;
    Vector Nmr;
    Vector Nlc;
    Vector Nfc;
    double kE;
    double cpl;
    double cpf;
    double Eld;
    double Efd;
    Vector Nld;
    Matrix DN;
    Vector DE;
    Vector DQ;
    // Growth curve parameters:
    struct {
        double kL;
        double kF;
    } Growthl;
    struct {
        double WmaxL;
        double aL;
```

```
        double tmaxL;
        double WmaxF;
        double aP;
        double tmaxF;
        double gestation;
        bool PhysAge;
    ] Growth2;
    int GrowthCurve;
    // Variables for graphical output:
    double Wb, We, cpb;
    double dWlt, dWft, dWb;
    double QP_digestion, QP_maintenance, QP_heat, QP_lean, QP_fat;
    double QPT, QPT_digestion, QPT_maintenance, QPT_heat, QPT_lean, QPT_fat;
    double dWltT, dWftT, dWltPT, dWftPT;
};

extern TMetabolism Metabolism;
//---------------------------------------------------------------------------
#endif
```

SwineSimMetabolism.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "SwineSim.h"
#include "SwineSimMetabolism.h"
//---------------------------------------------------------------------------
TMetabolism Metabolism;

// Description of all variables that are transfered between disk file,
// memory and GUI:
TVariableDescription TDataTransfer::MetabolismDescription [52];

static const System::AnsiString Section = "AnimalMetabolism";

#define SMD(i,ADDRESS,EDIT,NAME,DEFAULT) (          \
    MetabolismDescription[i].Address = (ADDRESS); \
    MetabolismDescription[i].Edit = (EDIT);       \
    MetabolismDescription[i].Name = (NAME);       \
    MetabolismDescription[i].Default = (DEFAULT); \
)

void TDataTransfer::SetMetabolismDescription (void) {
    SMD (0, &(Metabolism.InitialFatLeanRatio),MainWindow->MetFatLeanRatio,"InitialFatLeanRatio",0.7);
    SMD (1, &(Metabolism.kFL), MainWindow->kFL, "MinimumFatLeanRatio", 0.337);
    SMD (2, &(Metabolism.kWT), MainWindow->kWT, "WaterLeanRatio", 2.78);
    SMD (3, &(Metabolism.kEB), MainWindow->kEB, "EmptyBodyWeightFraction", 0.95);
    SMD (4, &(Metabolism.kF), MainWindow->kF, "LeanTissueEquivalentOfFatTissue", 0.5);
    SMD (5, &(Metabolism.EMN[POOL_PROTEIN]), MainWindow->EMP, "MetabolizableEnergy[Protein]", 16.5);
    SMD (6, &(Metabolism.EMN[POOL_FAT]), MainWindow->EML, "MetabolizableEnergy[Fat]", 39.6),
    SMD (7, &(Metabolism.EMN[POOL_CARBS]),MainWindow->EMG,"MetabolizableEnergy[Carbohydrates]",17.5);
    SMD (8, &(Metabolism.ENN[POOL_PROTEIN]), MainWindow->ENP, "NetEnergy[Protein]", 11.6);
    SMD (9, &(Metabolism.ENN[POOL_FAT]), MainWindow->ENL, "NetEnergy[Fat]", 39.6);
    SMD (10, &(Metabolism.ENN[POOL_CARBS]), MainWindow->ENG, "NetEnergy[Carbohydrates]", 17.5);
    SMD (11, &(Metabolism.Nlt[POOL_PROTEIN]), MainWindow->Plt, "LeanTissueComposition[Protein]",842);
    SMD (12, &(Metabolism.Nlt[POOL_FAT]), MainWindow->Llt, "LeanTissueComposition[Fat]", 0);
    SMD (13, &(Metabolism.Nlt[POOL_CARBS]),MainWindow->Glt,"LeanTissueComposition[Carbohydrates]",0);
    SMD (14, &(Metabolism.Nft[POOL_PROTEIN]), MainWindow->Pft, "FatTissueComposition[Protein]", 0);
    SMD (15, &(Metabolism.Nft[POOL_FAT]), MainWindow->Lft, "FatTissueComposition[Fat]", 1000);
    SMD (16, &(Metabolism.Nft[POOL_CARBS]), MainWindow->Gft,"FatTissueComposition[Carbohydrates]",0);
    SMD (17, &(Metabolism.Emin_), MainWindow->Emin_, "ThermoneutralMinPoolEnergy", 50);
    SMD (18, &(Metabolism.Emax_), MainWindow->Emax_, "ThermoneutralMaxPoolEnergy", 51);
    SMD (19, &(Metabolism.kv), MainWindow->kv, "CapacityOfDigestiveTract", 285);
    SMD (20, &(Metabolism.ri_), MainWindow->ri_, "ThermoneutralFeedIntakeRate", 0.87);
    SMD (21, &(Metabolism.rd), MainWindow->rd, "FeedDigestionRate", 0.4);
    SMD (22, &(Metabolism.Emr_), MainWindow->Emr_, "MaintenanceEnergyRequirement", 20);
    SMD (23, &(Metabolism.kr), MainWindow->kr, "IncreaseInMaintenanceEnergyByHeatStress", 0);
    SMD (24, &(Metabolism.Nmr[POOL_PROTEIN]),MainWindow->Pmr,"MaintenanceRequirement[Protein]",0.05);
    SMD (25, &(Metabolism.Nmr[POOL_FAT]), MainWindow->Lmr, "MaintenanceRequirement[Fat]", 0);
    SMD (26,&(Metabolism.Nmr[POOL_CARBS]),MainWindow->Gmr,"MaintenanceRequirement[Carbohydrates]",0);
    SMD (27, &(Metabolism.Nlc[POOL_PROTEIN]), MainWindow->Plc,
       "NutrientsFromLeanCatabolism[Protein]", 842);
    SMD (28, &(Metabolism.Nlc[POOL_FAT]), MainWindow->Llc, "NutrientsFromLeanCatabolism[Fat]", 0);
    SMD (29, &(Metabolism.Nlc[POOL_CARBS]), MainWindow->Glc,
       "NutrientsFromLeanCatabolism[Carbohydrates]", 0);
    SMD (30,&(Metabolism.Nfc[POOL_PROTEIN]),MainWindow->Pfc,"NutrientsFromFatCatabolism[Protein]",0);
    SMD (31, &(Metabolism.Nfc[POOL_FAT]), MainWindow->Lfc, "NutrientsFromFatCatabolism[Fat]", 1000);
    SMD (32, &(Metabolism.Nfc[POOL_CARBS]), MainWindow->Gfc,
       "NutrientsFromFatCatabolism[Carbohydrates]", 0);
```

```
    SMD (33, &(Metabolism.kE), MainWindow->kE, "EnergyCatabolizedFromFat", 0.5);
    SMD (34, &(Metabolism.cpl), MainWindow->cpl, "SpecificHeatofLeanTissue", 3.47);
    SMD (35, &(Metabolism.cpf), MainWindow->cpf, "SpecificHeatofFatTissue", 1.88);
    SMD (36, &(Metabolism.qmax), MainWindow->qmax, "MaximumAdditionalHeatProduction", 5.56);
    SMD (37, &(Metabolism.Eld), MainWindow->Eld, "LeanDepositionEnergyRequirement", 3090);
    SMD (38, &(Metabolism.Efd), MainWindow->Efd, "FatDepositionEnergyRequirement", 53500);
    SMD (39, &(Metabolism.Nld[POOL_PROTEIN]), MainWindow->Pld,
      "LeanDepositionRequirement[Protein]", 936);
    SMD (40, &(Metabolism.Nld[POOL_FAT]), MainWindow->Lld, "LeanDepositionRequirement[Fat]", 0);
    SMD (41, &(Metabolism.Nld[POOL_CARBS]), MainWindow->Gld,
      "LeanDepositionRequirement[Carbohydrates]", 0);
    SMD (42, &(Metabolism.Growth1.kL), MainWindow->Growth1_kL, "GrowthCurve1.kL", 0.204);
    SMD (43, &(Metabolism.Growth1.kF), MainWindow->Growth1_kF, "GrowthCurve1.kF", 0.150);
    SMD (44, &(Metabolism.Growth2.WmaxL), MainWindow->Growth2_WmaxL, "GrowthCurve2.WmaxL", 43.75);
    SMD (45, &(Metabolism.Growth2.aL), MainWindow->Growth2_aL, "GrowthCurve2.aL", 3.068);
    SMD (46, &(Metabolism.Growth2.tmaxL), MainWindow->Growth2_tmaxL, "GrowthCurve2.tmaxL", 280);
    SMD (47, &(Metabolism.Growth2.WmaxF), MainWindow->Growth2_WmaxF, "GrowthCurve2.WmaxF", 75);
    SMD (48, &(Metabolism.Growth2.aF), MainWindow->Growth2_aF, "GrowthCurve2.aF", 3.8);
    SMD (49, &(Metabolism.Growth2.tmaxF), MainWindow->Growth2_tmaxF, "GrowthCurve2.tmaxF", 325);
    SMD (50, &(Metabolism.Growth2.gestation), MainWindow->Growth2_gestation,
       "GrowthCurve2.gestation", 114);
    SMD (51, NULL, NULL, NULL, 0);
}

const double TDataTransfer::DigestionMatrixDefaults [POOL_NUTRIENTS + 2][FEED_NUTRIENTS] = {
    { 0.912, -0.028, -0.028, -0.028, 0 },
    { 0, 0.8, 0, 0, 0 },
    { 0, 0, 0.8, 0.336, 0 },
    { 0.2, 0.2, 0.2, 0.2, 0 },
    { 0, 0, 0, 0.432, 0 }
};

// Initializes other GUI parameters related to environmental variables:
void TMainWindow::InitMetaTab (void) {
    DigestionGrid->DefaultColWidth = DigestionGrid->Width / DigestionGrid->ColCount - 1;
    DigestionGrid->DefaultRowHeight = DigestionGrid->Height / DigestionGrid->RowCount - 1;
    DigestionGrid->ColWidths[0] = DigestionGrid->Width -
        (DigestionGrid->DefaultColWidth + 1) * (DigestionGrid->ColCount - 1) - 5;
    DigestionGrid->RowHeights[0] = DigestionGrid->Height -
        (DigestionGrid->DefaultRowHeight + 1) * (DigestionGrid->RowCount - 1) - 5;
    DigestionGrid->Cells[FEED_PROTEIN+1][0] = "       Feed Protein";
    DigestionGrid->Cells[FEED_FAT+1][0] = "       Feed Fat";
    DigestionGrid->Cells[FEED_CARBS+1][0] = "Feed Carbohydrates";
    DigestionGrid->Cells[FEED_FIBER+1][0] = "       Feed Fiber";
    DigestionGrid->Cells[FEED_VOLUME+1][0] = "       Feed Volume";
    DigestionGrid->Cells[0][POOL_PROTEIN+1] = "  Digested Protein";
    DigestionGrid->Cells[0][POOL_FAT+1] = "       Digested Fat";
    DigestionGrid->Cells[0][POOL_CARBS+1] = "Dig. Carbohydrates";
    DigestionGrid->Cells[0][POOL_NUTRIENTS+1] = "Energy Req. (kJ/g)";
    DigestionGrid->Cells[0][POOL_NUTRIENTS+2] = " Heat Prod. (kJ/g)";

    GrowthGraph->GraphTitle = "Growth curves";
    GrowthGraph->LeftTitle = "Weight\n(kg)";
    GrowthGraph->NumSets = 3;
    GrowthGraph->ThisSet = 1;
       GrowthGraph->LegendText = "Body\nweight";
       SimGraph->ColorData = gphCyan;
       SimGraph->PatternData = 0;
    GrowthGraph->ThisSet = 2;
       GrowthGraph->LegendText = "Lean\ntissue";
       GrowthGraph->ColorData = gphLightRed;
       GrowthGraph->PatternData = 2;
    GrowthGraph->ThisSet = 3;
       GrowthGraph->LegendText = "Fat\ntissue";
       GrowthGraph->ColorData = gphBrown;
       GrowthGraph->PatternData = 0;
    GrowthGraph->NumPoints = MAX_AGE * 2 + 1;
    GrowthGraph->LabelEvery = MAX_AGE / 5;
    GrowthGraph->TickEvery = MAX_AGE / 10;
    for (short i = 0; i <= MAX_AGE; i += (short) 20) {
       GrowthGraph->ThisPoint = (short) (i * 2 + 1);
       GrowthGraph->LabelText = i;
    }
    DataTransfer.SetMetabolismDescription ();
}

// Reads all metabolism variables from the disk file to memory:
void TDataTransfer::ReadMetabolismFromFile (TIniFile * IniFile) {
    int i, j;
    System::AnsiString VarName;
    TVariableDescription * VarPtr;
    for (VarPtr = MetabolismDescription; VarPtr->Address; VarPtr ++)
       *(VarPtr->Address) = atof (IniFile->ReadString (Section,
           VarPtr->Name, VarPtr->Default).c_str());
    Metabolism.GrowthCurve = IniFile->ReadInteger (Section, "GrowthCurve", 2);
    Metabolism.Growth2.PhysAge = IniFile->ReadBool (Section, "GrowthCurve2.PhysAge", true);
    VarName = "Digestion";
```

```
    for (i = 0; i < POOL_NUTRIENTS; i ++) {
        for (j = 0; j < FEED_NUTRIENTS; j ++) {
            Metabolism.DN[i][j] = atof (IniFile->ReadString (Section,
                VarName + "Matrix[" + i + "][" + j + "]", DigestionMatrixDefaults[i][j]).c_str());
        }
    }
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        Metabolism.DE[j] = atof (IniFile->ReadString (Section,
            VarName + "Energy[" + j + "]", DigestionMatrixDefaults[POOL_NUTRIENTS][j]).c_str());
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        Metabolism.DQ[j] = atof (IniFile->ReadString (Section,
            VarName + "Heat[" + j + "]", DigestionMatrixDefaults[POOL_NUTRIENTS+1][j]).c_str());
}

// Writes all metabolism variables from memory to the disk file:
void TDataTransfer::WriteMetabolismToFile (TIniFile * IniFile) {
    int i, j;
    System::AnsiString VarName;
    TVariableDescription * VarPtr;
    for (VarPtr = MetabolismDescription; VarPtr->Address; VarPtr ++)
        IniFile->WriteString (Section, VarPtr->Name, *(VarPtr->Address));
    IniFile->WriteInteger(Section, "GrowthCurve", Metabolism.GrowthCurve);
    IniFile->WriteBool(Section, "GrowthCurve2.PhysAge", Metabolism.Growth2.PhysAge);
    VarName = "Digestion";
    for (i = 0; i < POOL_NUTRIENTS; i ++) {
        for (j = 0; j < FEED_NUTRIENTS; j ++)
            IniFile->WriteString (Section, VarName + "Matrix["+i+"]["+j+"]", Metabolism.DN[i][j]);
    }
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        IniFile->WriteString (Section, VarName+"Energy["+j+"]", Metabolism.DE[j]);
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        IniFile->WriteString (Section, VarName+"Heat["+j+"]", Metabolism.DQ[j]);
}

// Reads all metabolism variables from the GUI to memory:
void TDataTransfer::ReadMetabolismFromScreen (void) {
    int i, j;
    TVariableDescription * VarPtr;
    for (VarPtr = MetabolismDescription; VarPtr->Address; VarPtr ++)
        *(VarPtr->Address) = atof (VarPtr->Edit->Text.c_str());
    if (MainWindow->GrowthRb1->Checked) Metabolism.GrowthCurve = 1;
    else if (MainWindow->GrowthRb2->Checked) Metabolism.GrowthCurve = 2;
    else if (MainWindow->GrowthRb3->Checked) Metabolism.GrowthCurve = 3;
    else Metabolism.GrowthCurve = 0;
    Metabolism.Growth2.PhysAge = MainWindow->Growth2_PhysAge->Checked;
    for (i = 0; i < POOL_NUTRIENTS; i ++) {
        for (j = 0; j < FEED_NUTRIENTS; j ++)
            Metabolism.DN[i][j] = atof (MainWindow->DigestionGrid->Cells[j+1][i+1].c_str());
    }
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        Metabolism.DE[j] = atof (MainWindow->DigestionGrid->Cells[j+1][POOL_NUTRIENTS+1].c_str());
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        Metabolism.DQ[j] = atof (MainWindow->DigestionGrid->Cells[j+1][POOL_NUTRIENTS+2].c_str());
}

// Writes all metabolism variables from memory to the GUI:
void TDataTransfer::WriteMetabolismToScreen (void) {
    int i, j;
    TVariableDescription * VarPtr;
    for (VarPtr = MetabolismDescription; VarPtr->Address; VarPtr ++)
        VarPtr->Edit->Text = *(VarPtr->Address);
    switch (Metabolism.GrowthCurve) {
        case 1: MainWindow->GrowthRb1->Checked = true; break;
        case 2: MainWindow->GrowthRb2->Checked = true; break;
        case 3: MainWindow->GrowthRb3->Checked = true; break;
    }
    MainWindow->Growth2_PhysAge->Checked = Metabolism.Growth2.PhysAge;
    for (i = 0; i < POOL_NUTRIENTS; i ++) {
        for (j = 0; j < FEED_NUTRIENTS; j ++)
            MainWindow->DigestionGrid->Cells[j+1][i+1] = Metabolism.DN[i][j];
    }
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        MainWindow->DigestionGrid->Cells[j+1][POOL_NUTRIENTS+1] = Metabolism.DE[j];
    for (j = 0; j < FEED_NUTRIENTS; j ++)
        MainWindow->DigestionGrid->Cells[j+1][POOL_NUTRIENTS+2] = Metabolism.DQ[j];
}

void TMetabolism::StartSimulation (double Weight, double Age, double TimeStep) {
    Wlt = Weight * kEB / (1 + kWT + InitialFatLeanRatio);
    Wft = Wlt * InitialFatLeanRatio;
    Wb = (Wlt * (1 + kWT) + Wft) / kEB; // body weight (kg)
    We = Wb - (1 - kF) * Wft / kEB; // equivalent body weight (kg)
    Np = DN * FeedingProgram.Get_Nf ();
    if ((EMN * Np) <= 0)
        throw ("Feed has no energy");
    Np = Np * (Emax_ * We / (EMN * Np));
    AnimalIsEating = false;
```

```
    Ni = 0;
    Ni[FEED_VOLUME] = ROUND_OFF;
    SimulateStep (TimeStep, Age, true);
    QPT = QPT_digestion = QPT_maintenance = QPT_heat = QPT_lean = QPT_fat = 0;
    dWltT = dWftT = dWltPT = dWftPT = 0;
}

void TMetabolism::SimulateStep (double TimeStep, int Age, bool ConstantWeight) {
    int i;
    double x;
    double OldLean = Wlt;
    double OldFat = Wft;
    Wb = (Wlt * (1 + kWT) + Wft) / kEB; // body weight (kg)
    We = Wb - (1 - kF) * Wft / kEB; // equivalent body weight (kg)
    QP = 0; // heat production in the time step (kJ)
    double Ra = FeedingProgram.Get_Ra (); // available feed (kg)
    Vector Nf = FeedingProgram.Get_Nf (); // vector of feed composition
    if (We <= 0)
        throw ("Equivalent body weight is zero or negative");
    double Ec = (ENN * Np) / We;
        // energy concentration in the nutrient pool (kJ/kg We)
    double Emin = HeatBalance.HSlevel (Emin_, 0);
        // minimum energy concentration level (kJ/kg Wb)
    double Emax = HeatBalance.HSlevel (Emax_, 0);
        // maximum energy concentration level (kJ/kg Wb)
    bool XF = FeedingProgram.Get_XF(); // feeding stimulus;
    if (XF || (Ec <= Emin)) AnimalIsEating = true;
    if (Ec >= Emax) AnimalIsEating = false;
    double Vmax = kv * pow (We, 0.75);
        // maximum capacity of the digestive tract (cm3)
    double Vi = Ni[FEED_VOLUME];
        // volume of the feed in the digestive tract (cm3)
    double ri = HeatBalance.HSlevel (ri_, 0) * AnimalIsEating * (Ra > ROUND_OFF);
        // rate of feed intake (1/h)
    double vf = Nf[FEED_VOLUME];
        // specific volume of the feed (cm3/kg)
    if (ri < 0)
        throw ("Rate of feed intake is negative");
    if (rd <= 0)
        throw ("Rate of digestion is zero or negative");
    x = (1 - exp (- (ri + rd) * TimeStep)) * (Vmax * ri / (ri + rd) - Vi);
    if (vf <= 0)
        throw ("Specific volume of feed is zero or negative");
    Ri = (rd * Vmax * TimeStep + x) * ri / ((ri + rd) * vf);
        // feed intake in the time step (kg)
    if (Vi + Ri * vf <= 0)
        throw ("Volume of ingested feed is zero or negative");
    double Rd = (ri * Vmax * TimeStep - x) * rd / ((ri + rd) * (Vi + Ri * vf));
        // fraction of feed digested in the time step
    if (Ri > Ra) Ri = Ra;
    Ni = Ni + Ri * Nf;
    Np = Np + Rd * (DN * Ni);
    double ENd = Rd * (DE * Ni); // net energy requirement for digestion (kJ)
    QP_digestion = Rd * (DQ * Ni);
    Ni = (1 - Rd) * Ni;
    double Emr = HeatBalance.HSrise (Emr_, kr); // energy requirement for maintenance (kJ/h.kg^0.75)
    QP_maintenance = (EMN * Nmr) * TimeStep * pow (We, 0.75);
    Np = Np - Nmr * TimeStep * pow (We, 0.75);
    double WlcN = 0; // lean tissue catabolized for nutrients (kg)
    for (int i = 0; i < POOL_NUTRIENTS; i ++) {
        if (Np[i] < 0) {
            if (Nlc[i] > 0) {
                if (WlcN < - Np[i] / Nlc[i])
                    WlcN = - Np[i] / Nlc[i];
            } else {
                if (Np[i] > - ROUND_OFF)
                    Np [i] = 0;
                else
                    throw ("Defficiency of a nutrient with no reserves");
            }
        }
    }
    if (WlcN > 0) {
        QP_maintenance += WlcN * EMN * (Nlt - Nlc);
        Np = Np + WlcN * Nlc;
        Wlt -= WlcN;
    }
    double ENm = (Emr - ENN * Nmr) * TimeStep * pow (We, 0.75) + ENd;
        // net energy requirement for maintenance (kJ)
    double ENp = ENN * Np; // net energy in the nutrient pool (kJ)
    if (ENm >= ENp) {
        QP_digestion += ENN * Np * (ENd / ENm);
        QP_maintenance += EMN * Np * (1 - ENd / ENm);
        Np = 0;
    } else if (ENm < ENp) {
        if (ENm < 0)
            throw ("Net energy requirement for maintenance is negative");
```

```
    QP_digestion += ENN * Np * (ENd / ENp);
    QP_maintenance += ENN * Np * ((ENm - ENd) / ENp);
    Np = Np - Np * (ENm / ENp);
} else {
    if (ENN * Nfc <= 0)
        throw ("Net energy from fat catabolism is zero or negative");
    if (ENN * Nlc <= 0)
        throw ("Net energy from lean catabolism is zero or negative");
    double WfcE = (ENm - ENp) * kE / (ENN * Nfc); // excess fat tissue catabolized for energy (kg)
    if (WfcE > Wft - kFL * Wlt)
        WfcE = Wft - kFL * Wlt;
    x = (ENm - ENp - WfcE * ENN * Nfc) / (ENN * (Wlt * Nlc + (Wft - WfcE) * Nfc));
    double WlcE = Wlt * x; // lean tissue catabolized for energy (kg)
    WfcE += (Wft - WfcE) * x;
    QP_digestion += ENN * (Np + WlcE * Nlt + WfcE * Nft) * (ENd / ENm);
    QP_maintenance += ENN * (Np + WlcE * Nlt + WfcE * Nft) * (1 - ENd / ENm);
    Np = 0;
    Wlt -= WlcE;
    Wft -= WfcE;
}
cpb = (Wlt*cpl + Wft*cpf + Wlt*kWT*HeatBalance.cpw) / (Wlt*(1+kWT) + Wft);
    // specific heat of the animal's body (kJ/kg.C)
double EMq = HeatBalance.CStemp() * cpb * Wb;
    // additional metabolizable energy for heat production (kJ)
x = qmax * pow (We, 0.75) * TimeStep * 3.6;
if (EMq > x) EMq = x;
double EMp = EMN * Np; // metabolizable energy in the nutrient pool (kJ)
QP_heat = 0;
if (EMq == EMp) {
    QP_heat += EMq;
    Np = 0;
} else if (EMq < EMp) {
    if (EMq < 0)
        throw ("Energy requirement for additional heat is negative");
    QP_heat += EMq;
    Np = Np - Np * (EMq / EMp);
} else {
    if (EMN * Nfc <= 0)
        throw ("Metabolizable energy from fat catabolism is zero or negative");
    if (EMN * Nlc <= 0)
        throw ("Metabolizable energy from lean catabolism is zero or negative");
    double WfcQ = (EMq - EMp) * kE / (EMN * Nfc);
        // excess fat tissue catabolized for additional heat (kg)
    if (WfcQ > Wft - kFL * Wlt)
        WfcQ = Wft - kFL * Wlt;
    x = (EMq - EMp - WfcQ * EMN * Nfc) / (EMN * (Wlt * Nlc + (Wft - WfcQ) * Nfc));
    double WlcQ = Wlt * x;
        // lean tissue catabolized for additional heat (kg)
    WfcQ += (Wft - WfcQ) * x;
    QP_heat += EMq;
    Np = 0;
    Wlt -= WlcQ;
    Wft -= WfcQ;
}
double rld = LeanGrowthCurve (Age, Wlt, Wft) / 24;
    // maximum rate of lean tissue deposition (kg/h)
double rfd = FatGrowthCurve (Age, Wlt, Wft) / 24;
    // maximum rate of excess fat tissue deposition (kg/h)
QP_lean = 0;
QP_fat = 0;
if (Eld <= 0)
    throw ("Energy requirement for lean deposition is zero or negative");
double Wld = (ENN * Np) / (Eld + kFL * Efd);
    // lean tissue deposition (kg)
if (Wld > rld * TimeStep) Wld = rld * TimeStep;
for (i = 0; i < POOL_NUTRIENTS; i ++) {
    if (Nld[i] > 0)
        if (Wld > Np[i] / Nld[i])
            Wld = Np[i] / Nld[i];
}
if (Wld > 0) {
    Np = Np - Wld * Nld;
    x = (ENN * Np) / (ENN * Np);
    QP_lean += Wld * (Eld * x - ENN * Nlt);
    QP_fat += Wld * kFL * (Eld * x - ENN * Nft);
    Np = Np - Wld * (((Eld + kFL * Efd) / (ENN * Np)) * Np - Nld);
    Wlt += Wld;
    Wft += kFL * Wld;
}
if (Efd <= 0)
    throw ("Energy requirement for fat deposition is zero or negative");
double Wfd = (ENN * Np) / Efd;
    // excess fat tissue deposition (kg)
x = rfd * TimeStep - kFL * Wld;
if (Wfd > x) Wfd = x;
if (Wfd > 0) {
    x = Efd / (ENN * Np);
```

```
        QP_fat += Wfd * EMN * (x * Np - Nft);
        Np = Np - Wfd * x * Np;
        Wft += Wfd;
    }
    QP = QP_digestion + QP_maintenance + QP_heat + QP_lean + QP_fat;
    dWlt = (Wlt - OldLean) * 24 / TimeStep;
    dWft = (Wft - OldPat) * 24 / TimeStep;
    dWb = ((Wlt * (1 + kWT) + Wft) / kEB - Wb) * 24 / TimeStep;
    if (ConstantWeight) {
        Wlt = OldLean;
        Wft = OldPat;
    } else {
        Wb = (Wlt * (1 + kWT) + Wft) / kEB; // body weight (kg)
        We = Wb - (1 - kF) * Wft / kEB; // equivalent body weight (kg)
    }
    QPT += QP;
    QPT_digestion += QP_digestion;
    QPT_maintenance += QP_maintenance;
    QPT_heat += QP_heat;
    QPT_lean += QP_lean;
    QPT_fat += QP_fat;
    dWltT += dWlt * TimeStep / 24;
    dWftT += dWft * TimeStep / 24;
    dWltPT += rld * TimeStep;
    dWftPT += rfd * TimeStep;
}

// Returns the growth rate for lean tissue (kg/day), using the selected growth
// curve; Growth Curve 1 returns a constant, independent of age and weight;
// Growth Curve 2 returns the growth rate as a function of physiological age;
// the physiological age is calculated as a function of lean tissue mass;
// if LeanMass is zero (physiological age would be zero), Age is used as the
// physiological age
double TMetabolism::LeanGrowthCurve (int Age, double LeanMass, double FatMass) {
    double t, m, A;
    switch (GrowthCurve) {
        case 1: return (Growth1.kL);
        case 2:
        case 3:
            t = ((LeanMass < 0) || ! Growth2.PhysAge ? Age + Growth2.gestation :
                Growth2.tmaxL * pow (- log (1 - LeanMass / Growth2.WmaxL)
                * Growth2.aL / (Growth2.aL - 1), 1 / Growth2.aL));
            m = (Growth2.aL - 1) / (Growth2.aL * pow (Growth2.tmaxL, Growth2.aL));
            A = Growth2.WmaxL * Growth2.aL * m;
            return (A * pow (t, Growth2.aL - 1) * exp (-m * pow (t, Growth2.aL)));
        default: return (0);
    }
}

double TMetabolism::FatGrowthCurve (int Age, double LeanMass, double FatMass) {
    double t, m, A;
    switch (GrowthCurve) {
        case 1: return (Growth1.kF);
        case 2:
        case 3:
            t = ((LeanMass < 0) || ! Growth2.PhysAge ? Age + Growth2.gestation :
                Growth2.tmaxL * pow (- log (1 - LeanMass / Growth2.WmaxL)
                * Growth2.aL / (Growth2.aL - 1), 1 / Growth2.aL));
            m = (Growth2.aF - 1) / (Growth2.aF * pow (Growth2.tmaxF, Growth2.aF));
            A = Growth2.WmaxF * Growth2.aF * m;
            return (A * pow (t, Growth2.aF - 1) * exp (-m * pow (t, Growth2.aF)));
        default: return (0);
    }
}

void TMainWindow::UpdateGrowthGraph (void) {
    double Wlt, Wft;
    double kL, kF;
    double WmaxL, aL, tmaxL, WmaxF, aF, tmaxF, gestation, mL, mF;
    int GrowthCurve;
    if (GrowthKb1->Checked) GrowthCurve = 1;
    else if (GrowthKb2->Checked) GrowthCurve = 2;
    else if (GrowthKb3->Checked) GrowthCurve = 3;
    else GrowthCurve = 0;
    double StartAge = atof (SimStartAge->Text.c_str());
    double StartWeight = atof (SimStartWeight->Text.c_str());
    double FatLeanRatio = atof (MetFatLeanRatio->Text.c_str());
    double kWT = atof (TMainWindow::kWT->Text.c_str());
    double kEB = atof (TMainWindow::kEB->Text.c_str());
    double StartWlt = StartWeight * kEB / (1 + FatLeanRatio + kWT);
    double StartWft = StartWeight * kEB * FatLeanRatio / (1 + FatLeanRatio + kWT);
    switch (GrowthCurve) {
        case 1:
            kL = atof (Growth1_kL->Text.c_str());
            kF = atof (Growth1_kF->Text.c_str());
            Wlt = StartWlt - StartAge * kL;
            Wft = StartWft - StartAge * kF;
```

```
        break;
        case 2:
        case 3:
            WmaxL = atof (Growth2_WmaxL->Text.c_str());
            aL = atof (Growth2_aL->Text.c_str());
            tmaxL = atof (Growth2_tmaxL->Text.c_str());
            WmaxF = atof (Growth2_WmaxF->Text.c_str());
            aF = atof (Growth2_aF->Text.c_str());
            tmaxF = atof (Growth2_tmaxF->Text.c_str());
            gestation = atof (Growth2_gestation->Text.c_str());
            mL = (aL - 1) / (aL * pow (tmaxL, aL));
            mF = (aF - 1) / (aF * pow (tmaxF, aF));
            Wlt = WmaxL * (1 - exp (- mL * pow (gestation, aL)));
            Wft = WmaxF * (1 - exp (- mF * pow (gestation, aF)));
        break;
        default:
            Wlt = Wft = 0;
        break;
    }
    for (short i = 1; i <= GrowthGraph->NumPoints; i ++) {
        GrowthGraph->ThisPoint = i;
        GrowthGraph->ThisSet = 1;
        GrowthGraph->GraphData = (Wft + Wlt * (1 + kWT)) / kEB;
        GrowthGraph->ThisSet = 2;
        GrowthGraph->GraphData = Wlt;
        GrowthGraph->ThisSet = 3;
        GrowthGraph->GraphData = Wft;
        switch (GrowthCurve) {
            case 1:
                Wlt += kL / 2;
                Wft += kF / 2;
            break;
            case 2:
                Wlt = WmaxL * (1 - exp (- mL * pow (gestation + i / 2, aL)));
                Wft = WmaxF * (1 - exp (- mF * pow (gestation + i / 2, aF)));
            break;
            case 3:
                Wft += Metabolism.FatGrowthCurve (0, Wlt, 0) / 2;
                Wlt += Metabolism.LeanGrowthCurve (0, Wlt, 0) / 2;
            break;
            default:
                Wlt = 0;
                Wft = 0;
            break;
        }
    }
    GrowthGraph->DrawMode = gphBlit;
}
```

SwineSimExpert.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimExpertH
#define SwineSimExpertH
//---------------------------------------------------------------------------
class Expert; // general class for general control of the expert system
class Parameter; // class describing parameters
class Rule; // base class from which rules are derived

struct ParamPtr { // this structure is a list of parameters
    Parameter * param; // points to the structure describing the parameter
    ParamPtr * next; // points to the next element in the list
};

struct RulePtr { // this structure is a list of rules
    Rule * rule; // points to the structure describing the rule
    RulePtr * next; // points to the next element in the list
};
//---------------------------------------------------------------------------
extern Expert Ex; // define object 'Ex', to implement class Expert
//---------------------------------------------------------------------------
class Expert {
public:
    Expert (); // constructor initializes parameter and rule lists
    ~Expert (); // destructor deallocates memory used by parameter and rule lists
    void UseParam (Parameter * P); // adds a parameter to the parameter list
    void UseRule (Rule * R); // adds a rule to the rule list
    void Log (System::AnsiString LogLine); // writes the LogLine string to the log file
    void Indent (void) { Margin ++; };
    void Outdent (void) { if (Margin) Margin --; };
    void StartExpert (TStrings * LogStringList);
```

```
        // start the expert system, putting the output in LogStringList
    void ExecuteLine (System::AnsiString Line);
        // execute a line of text in the form 'parameter=value'
    void FinishExpert (void);
        // finish the expert system execution, disabling further output
private:
    TStrings * LogList; // string list where the output is logged
    ParamPtr * FirstP; // first parameter in the parameter list
    ParamPtr * LastP; // last parameter in the parameter list
    RulePtr * FirstR; // first rule in the rule list
    RulePtr * LastR; // last rule in the rule list
    int Margin; // level of indentation
};
//---------------------------------------------------------------------------
class Parameter {
public:
    Parameter (const char * ParamName, double DefaultValue = 0);
        // constructor sets the parameter name and its default value
    ~Parameter (); // destructor deallocates memory
    void UseRule (Rule * R); // adds a rule to the rule list
    const char * GetName (void) { return (Name); }; // name of the parameter
    void ClearValue (void) { Value = Default; }
        // Sets the value of a parameter, back to its default value:
    void Set (double value);
    double Get (void) { return (Value); };
private:
    RulePtr * FirstR; // first rule in the rule list
    RulePtr * LastR; // last rule in the rule list
    const char * Name; // name of the parameter
    double Default; // default value for the parameter
    double Value; // current value for the parameter
};
//---------------------------------------------------------------------------
class Rule {
public:
    Rule () { Executed = 0; Ex.UseRule (this); };
        // constructor initializes variables and registers rule
    const char * GetName (void){ return (Name); };
    int GetExecuted (void){ return (Executed); };
    void SetExecuted (void){ Executed = 1; };
    void ClearExecuted (void){ Executed = 0; };
    virtual int Premise (void) = 0; // must be implemented in the derived class
    virtual void Action (void) = 0; // must be implemented in the derived class
protected:
    const char * Name; // name of the rule
private:
    int Executed; // 0 if the rule was not yet executed; 1 if it already was
};
//---------------------------------------------------------------------------
#endif
```

SwineSimExpert.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include <stdio.h>
#include <string.h>
#include "SwineSim.h"
#include "SwineSimExpert.h"

//---------------------------------------------------------------------------
Expert Ex; // define object 'Ex', to implement class Expert
//---------------------------------------------------------------------------

// Constructor for an expert system object:
Expert::Expert () {
   // Initialize list of parameters to empty:
   FirstP = LastP = 0;
   // Initialize list of rules to empty:
   FirstR = LastR = 0;
   // Set the indentation level to zero:
   Margin = 0;
   // Set the pointer to the log string list to null:
   LogList = 0;
}

// Destructor for an expert system object:
Expert::~Expert () {
   // Delete all parameters from list of parameters:
```

```
    while (FirstP) {
        ParamPtr * P = FirstP->next;
        delete FirstP;
        FirstP = P;
    }
    // Delete all rules from list of rules:
    while (FirstR) {
        RulePtr * R = FirstR->next;
        delete FirstR;
        FirstR = R;
    }
}

// Add a parameter to the expert system's list of parameters, so that the expert
// system can look for the parameter when processing the input file and write
// the parameter's value to the output file:
void Expert::UseParam (Parameter * P) {
    if (LastP) {
        LastP->next = new ParamPtr;
        LastP = LastP->next;
    } else {
        FirstP = new ParamPtr;
        LastP = FirstP;
    };
    LastP->param = P;
    LastP->next = 0;
};

// Add a rule to the expert system's list of rules, so that the rule's executed
// flag can be cleared before the expert system runs:
void Expert::UseRule (Rule * R) {
    if (LastR) {
        LastR->next = new RulePtr;
        LastR = LastR->next;
    } else {
        FirstR = new RulePtr;
        LastR = FirstR;
    };
    LastR->rule = R;
    LastR->next = 0;
};

// Add a line to the log:
void Expert::Log (System::AnsiString LogLine) {
    // Indent the proper number of spaces:
    for (int i = Margin; i; i --)
        LogLine = "   " + LogLine;
    // Write the line to the log string list:
    if (LogList)
        LogList->Append (LogLine);
}

// Begin the expert system execution; 'LogStringList' points to the string list
// that will accept the expert system output:
void Expert::StartExpert (TStrings * LogStringList) {
    // Set the value of the log string list to enable logging:
    LogList = LogStringList;
    // Empty the log:
    if (LogList)
        LogList->Clear();
    // Write a header to the log, to signal the start of the execution:
    Log ("=== SwineSim Expert System execution log ===");
    // Mark all rules as not yet having been executed:
    for (RulePtr * R = FirstR; R; R = R->next)
        R->rule->ClearExecuted ();
    // Set all parameters back to their default values:
    for (ParamPtr * P = FirstP; P; P = P->next)
        P->param->ClearValue ();
}

// Executes one line of the data base; the line should be in the form
// 'parameter=value', where 'parameter' is the name of the parameter
// being set and 'value' is the value it is being set to:
void Expert::ExecuteLine (System::AnsiString ExpertLine) {
    // Separate the parameter name from the parameter value (break on the '='):
    System::AnsiString LineParameter = ExpertLine.SubString (1,
        ExpertLine.Pos ("=")-1).Trim();
    System::AnsiString LineValue = ExpertLine.SubString (ExpertLine.Pos ("=")+1,
        ExpertLine.Length() - ExpertLine.Pos ("=")).Trim();
    // Get the value of the parameter:
    double value;
    sscanf (LineValue.c_str(), "%lf", & value);
    // Look for the parameter name in the list of parameters:
    for (ParamPtr * P = FirstP; P; P = P->next) {
        if (LineParameter == P->param->GetName()) {
            // When the parameter is found, set the value of the parameter
            // (possibly firing rules):
```

```
            P->param->Set (value);
            break;
        }
    }
}

// Finish the expert system execution:
void Expert::FinishExpert (void) {
    // Write a footer to the log file, to signal the end of the execution:
    Log ("=== Finished processing ===");
    // Disable output logging:
    LogList = 0;
}

//---------------------------------------------------------------------------

// Constructor for a parameter object:
Parameter::Parameter (const char * ParamName, double DefaultValue) {
    // Initialize list of rules to empty:
    FirstR = LastR = 0;
    // Set the name of the poarameter:
    Name = ParamName;
    // Set the default value of the parameter
    // (should indicate the parameter is not valid):
    Value = Default = DefaultValue;
    // Register the parameter in the expert system's parameter list:
    Ex.UseParam (this);
};

// Destructor for a parameter object:
Parameter::~Parameter () {
    // Delete all rules from the list of rules:
    while (FirstR) {
        RulePtr * R = FirstR->next;
        delete FirstR;
        FirstR = R;
    }
};

// Add a rule to the parameter's list of rules,
// indicating the execution of the rule depends on the parameter:
void Parameter::UseRule (Rule * R) {
    if (LastR) {
        LastR->next = new RulePtr;
        LastR = LastR->next;
    } else {
        FirstR = new RulePtr;
        LastR = FirstR;
    };
    LastR->rule = R;
    LastR->next = 0;
};

// Sets the value of a parameter, possibly executing rules:
void Parameter::Set (double value) {
    char buffer [256];
    // Set the value of the parameter to the new value:
    Value = value;
    // Write a line to the log file, indicating the action:
    sprintf (buffer, "Parameter '%s' set to %lf.", Name, value);
    Ex.Log (buffer);
    // Increase the indentation level in the log file:
    Ex.Indent ();
    // Test the rules that depend on the parameter for possible execution:
    for (RulePtr * R = FirstR; R; R = R->next) {
        // Test if the rule was not yet executed:
        if (! R->rule->GetExecuted ()) {
            // If the rule was not executed, write a line to the log file,
            // indicating the rule is being tested:
            // Test the premise of the rule:
            if (R->rule->Premise ()) {
                // If the premise is true, write a line to the log file,
                // indicating the rule is being executed:
                sprintf (buffer,
                    "Setting parameter '%s' caused rule '%s' to be executed.",
                    Name, R->rule->GetName ());
                Ex.Log (buffer);
                // Mark the rule as having been executed (to avoid infinite loops):
                R->rule->SetExecuted ();
                // Execute the rule's action (this may set other parameters):
                R->rule->Action ();
            }
        }
    }
    // Reduce the indentation level in the log file:
    Ex.Outdent ();
};
```

```
//---------------------------------------------------------------------------
```

SwineSimKnowledgeBase.h

```
//---------------------------------------------------------------------------
#ifndef SwineSimKnowledgeBaseH
#define SwineSimKnowledgeBaseH
//---------------------------------------------------------------------------
#endif
```

SwineSimKnowledgeBase.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "SwineSimExpert.h"
#include "SwineSimKnowledgeBase.h"
//---------------------------------------------------------------------------
// Definition of parameters:
//---------------------------------------------------------------------------
Parameter AdditionalHeat ("AdditionalHeat", -1);
Parameter LeanGrowthRate ("LeanGrowthRate", 2);
Parameter FatGrowthRate ("FatGrowthRate", 2);
Parameter GrowthRate ("GrowthRate", -99);
Parameter TimeCold ("TimeCold", -1);
Parameter TimeNeutral ("TimeNeutral", -1);
Parameter TimeEvaporating ("TimeEvaporating", -1);
Parameter TimeHot ("TimeHot", -1);
Parameter TempTimeCold ("TempTimeCold");
Parameter TempTimeNeutral ("TempTimeNeutral");
Parameter TempTimeEvaporating ("TempTimeEvaporating");
Parameter TempTimeHot ("TempTimeHot");
//---------------------------------------------------------------------------
// Definition of rules:
//---------------------------------------------------------------------------
class RuleS1:Rule {
public:
    RuleS1 () {
        Name = "Animal is cold";
        AdditionalHeat.UseRule (this);
        TimeCold.UseRule (this);
        TempTimeCold.UseRule (this);
    };
    int Premise (void) {
        return ((AdditionalHeat.Get() > 0.01) && (TimeCold.Get() > 0.01) && (TempTimeCold.Get() > 0));
    };
    void Action (void) {
        System::AnsiString Message;
        Message = AnsiString::FloatToStrF(AdditionalHeat.Get() * 100, AnsiString::sffFixed, 4, 2);
        Message += " % of the animal's heat production was for supplemental heat.";
        Ex.Log (Message);
        Message = "The environment was below the thermoneutral zone for ";
        Message += AnsiString::FloatToStrF(TimeCold.Get() * 100, AnsiString::sffFixed, 4, 2);
        Message += " % of the time.";
        Ex.Log (Message);
        Message = "During this time, average body temperature was ";
        Message += AnsiString::FloatToStrF (TempTimeCold.Get(), AnsiString::sffFixed, 4, 2);
        Message += "°C.";
        Ex.Log (Message);
        Ex.Log ("The system should be changed to heat the environment during these periods.");
    };
} RS1;
//---------------------------------------------------------------------------
class RuleS2:Rule {
public:
    RuleS2 () {
        Name = "Animal is hot";
        LeanGrowthRate.UseRule (this);
        FatGrowthRate.UseRule (this);
        TimeEvaporating.UseRule (this);
        TimeHot.UseRule (this);
        TempTimeEvaporating.UseRule (this);
        TempTimeHot.UseRule (this);
    };
```

```
    int Premise (void) {
        return ((
            ((LeanGrowthRate.Get() < 0.99) && (FatGrowthRate.Get() <= 1))
            ||
            ((FatGrowthRate.Get() < 0.95) && (LeanGrowthRate.Get() <=1))
        ) && (
            ((TimeEvaporating.Get() > 0.05) && (TimeHot.Get() > -1))
            ||
            ((TimeHot.Get() > 0.01) && (TimeEvaporating.Get() > -1))
        ) && (TempTimeEvaporating.Get() > 0) && (TempTimeHot.Get() > 0));
    };
    void Action (void) {
        System::AnsiString Message;
        Message = "The animal was growing lean tissue ";
        Message += AnsiString::FloatToStrF(100 * (1 - LeanGrowthRate.Get()), AnsiString::sffFixed, 4,
2);
        Message += " % below its potential and fat tissue ";
        Message += AnsiString::FloatToStrF(100 * (1 - FatGrowthRate.Get()), AnsiString::sffFixed, 4, 2);
        Message += " % below its potential.";
        Ex.Log (Message);
        Message = "The environment was above the thermoneutral zone for ";
        Message     +=     AnsiString::FloatToStrF((TimeEvaporating.Get()     +     TimeHot.Get())*     100,
AnsiString::sffFixed, 4, 2);
        Message += " % of the time.";
        Ex.Log (Message);
        Message = "During this time, average body temperature was ";
        Message += AnsiString::FloatToStrF ((TempTimeEvaporating.Get()
            * TimeEvaporating.Get() + TempTimeHot.Get() * TimeHot.Get())
            / (TimeEvaporating.Get()+ TimeHot.Get()), AnsiString::sffFixed, 4, 2);
        Message += "°C.";
        Ex.Log (Message);
        Message = AnsiString::FloatToStrF(TimeHot.Get()* 100, AnsiString::sffFixed, 4, 2);
        Message += " % of the time the animal was hyperthermic.";
        Ex.Log (Message);
        Message = "During this time, average body temperature was ";
        Message += AnsiString::FloatToStrF (TempTimeHot.Get(), AnsiString::sffFixed, 4, 2);
        Message += "°C.";
        Ex.Log (Message);
        Ex.Log ("The system should be changed to cool the environment during these periods.");
    };
} RS2;
```

Matrix.h

```
//---------------------------------------------------------------------------
#ifndef MatrixH
#define MatrixH

#include <mem.h>

class Vector;
class Matrix;
//---------------------------------------------------------------------------
class Vector {
    friend Matrix;
public:
    Vector () { // default constructor
        rows = 0;
        base = new double [1];
        * base = 0;
    };
    Vector (const Vector &V) { // copy constructor
        rows = V.rows;
        base = new double [rows ? rows : 1];
        if (rows)
            memmove (base, V.base, rows * sizeof (double));
        else
            * base = 0;
    };
    Vector (unsigned int nrows) { // fill-with-zeros constructor
        rows = nrows;
        base = new double [rows ? rows : 1];
        if (rows)
            memset (base, 0, rows * sizeof (double));
        else
            * base = 0;
    };
    Vector (unsigned int nrows, double * p) { // double-pointer constructor
        rows = nrows;
        base = new double [rows ? rows : 1];
```

```
        if (rows)
            memmove (base, p, rows * sizeof (double));
        else
            * base = 0;
    };
    virtual ~Vector () { // destructor
        delete [] base;
    };
    double& operator [] (unsigned int row) { // index operator
        return (base [row < rows ? row : 0]);
    };
    double& operator * () { // pointer operator
        return (* base);
    };
    Vector& operator = (const Vector &V) { // assign a vector to a vector
        if (this == &V) return (*this);
        delete [] base;
        rows = V.rows;
        base = new double [rows ? rows : 1];
        if (rows)
            memmove (base, V.base, rows * sizeof (double));
        else
            * base = 0;
        return (*this);
    }
    Vector& operator = (const double p) { // assign a double to all elements
        if (! rows)
            rows = 1;
        for (unsigned int n = 0; n < rows; n ++)
            base [n] = p;
        return (*this);
    };
    Vector operator - () const { // zero minus the vector
        Vector R (rows);
        for (unsigned int n = 0; n < rows; n ++)
            R.base [n] = - base [n];
        return (R);
    }
    Vector operator + (const Vector & V) const { // add two vectors
        Vector R (rows > V.rows ? rows : V.rows);
        for (unsigned int n = 0; n < R.rows; n ++)
            R.base [n] = (n < rows ? base [n] : 0) + (n < V.rows ? V.base [n] : 0);
        return (R);
    }
    Vector operator - (const Vector & V) const { // subtract two vectors
        Vector R (rows > V.rows ? rows : V.rows);
        for (unsigned int n = 0; n < R.rows; n ++)
            R.base [n] = (n < rows ? base [n] : 0) - (n < V.rows ? V.base [n] : 0);
        return (R);
    }
    double operator * (const Vector & V) const { // multiply vectors to generate a scalar
        double r = 0;
        for (unsigned int n = 0; n < (rows < V.rows ? rows : V.rows); n ++)
            r += base [n] * V.base [n];
        return (r);
    }
    friend Vector operator * (const Matrix & M, const Vector & V); // multiply a matrix by a vector
    friend Vector operator * (const Vector & V, const Matrix & M); // multiply a vector by a matrix
    friend Vector operator * (const Vector & V, const double s); // multiply a vector by a scalar
    friend Vector operator * (const double s, const Vector & V); // multiply a scalar by a vector
    friend Vector operator + (const Vector & V, const double s); // add a vector and a scalar
    friend Vector operator + (const double s, const Vector & V); // add a scalar and a vector
    friend Vector operator - (const Vector & V, const double s); // subtract a scalar from a vector
    friend Vector operator - (const double s, const Vector & V); // subtract a vector from a scalar
private:
    double *base;
    unsigned int rows;
};
//---------------------------------------------------------------------------
class Matrix {
public:
    Matrix () { // default constructor
        lins = cols = 0;
        base = new double [1];
        * base = 0;
    };
    Matrix (const Matrix &M) { // copy constructor
        lins = M.lins;
        cols = M.cols;
        base = new double [lins && cols ? lins * cols : 1];
        if (lins && cols)
            memmove (base, M.base, lins * cols * sizeof (double));
        else
            * base = 0;
    };
    Matrix (unsigned int lines, unsigned int columns = 0) { // fill-with-zeros constructor
        lins = lines;
```

```
    cols = columns;
    base = new double [lins && cols ? lins * cols : 1];
    if (lins && cols)
       memset (base, 0, lins * cols * sizeof (double));
    else
       * base = 0;
};
Matrix (unsigned int lines, unsigned int columns, double * p) { // double-pointer constructor
    lins = lines;
    cols = columns;
    base = new double [lins && cols ? lins * cols : 1];
    if (lins && cols)
       memmove (base, p, lins * cols * sizeof (double));
    else
       * base = 0;
};
virtual ~Matrix () { // destructor
    delete [] base;
};
double * operator [] (unsigned int lin) { // index operator
    return (base + (lin < lins ? lin * cols : 0));
}
double& operator * () { // pointer operator
    return (* base);
};
Matrix& operator = (const Matrix &M) { // assign a matrix to a matrix
    if (this == &M) return (*this);
    delete [] base;
    lins = M.lins;
    cols = M.cols;
    base = new double [lins && cols ? lins * cols : 1];
    if (lins && cols)
       memmove (base, M.base, lins * cols * sizeof (double));
    else
       * base = 0;
    return (*this);
}
Matrix& operator = (const double p) { // assign a double to all elements
    if (! (lins && cols))
       lins = cols = 1;
    for (unsigned int n = 0; n < lins * cols; n ++)
       base [n] = p;
    return (*this);
};
Matrix operator - () const { // zero minus the matrix
    Matrix R (lins, cols);
    for (unsigned int n = 0; n < lins * cols; n ++)
       R.base [n] = - base [n];
    return (R);
}
Matrix operator + (const Matrix & M) const { // add two matrices
    Matrix R ((lins > M.lins ? lins : M.lins), (cols > M.cols ? cols : M.cols));
    for (unsigned int i = 0; i < R.lins; i ++)
       for (unsigned int j = 0; j < R.cols; j ++)
          R.base [i * R.cols + j] = ((i < lins) && (j < cols) ? base [i * cols + j] : 0) +
                ((i < M.lins) && (j < M.cols) ? M.base [i * M.cols + j] : 0);
    return (R);
}
Matrix operator - (const Matrix & M) const { // subtract two matrices
    Matrix R ((lins > M.lins ? lins : M.lins),
              (cols > M.cols ? cols : M.cols));
    for (unsigned int i = 0; i < R.lins; i ++)
       for (unsigned int j = 0; j < R.cols; j ++)
          R.base [i * R.cols + j] = ((i < lins) && (j < cols) ? base [i * cols + j] : 0) -
                ((i < M.lins) && (j < M.cols) ? M.base [i * M.cols + j] : 0);
    return (R);
}
Vector Column (unsigned int i) {
    Vector R (lins);
    for (unsigned int n = 0; n < lins; n ++)
       R.base [n] = base [n * cols + (i < cols ? i : 0)];
    return (R);
}
Vector Line (unsigned int i) {
    Vector R (cols);
    memmove (R.base, base + (i < lins ? i : 0) * cols, cols * sizeof (double));
    return (R);
}
Matrix operator * (const Matrix & M) const; // multiply two matrices
friend Vector operator * (const Matrix & M, const Vector & V); // multiply a matrix by a vector
friend Vector operator * (const Vector & V, const Matrix & M); // multiply a vector by a matrix
friend Matrix operator * (const Matrix & M, const double s); // multiply a matrix by a scalar
friend Matrix operator * (const double s, const Matrix & M); // multiply a scalar by a matrix
friend Matrix operator + (const Matrix & M, const double s); // add a matrix and a scalar
friend Matrix operator + (const double s, const Matrix & M); // add a scalar and a matrix
friend Matrix operator - (const Matrix & M, const double s); // subtract a scalar from a matrix
friend Matrix operator - (const double s, const Matrix & M); // subtract a matrix from a scalar
```

```
private:
    double *base;
    unsigned int lins;
    unsigned int cols;
};
//---------------------------------------------------------------------------
#endif
```

Matrix.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Matrix.h"
//---------------------------------------------------------------------------
Matrix Matrix::operator * (const Matrix & M) const { // multiply two matrices
    Matrix R (lins, M.cols);
    for (unsigned int i = 0; i < R.lins; i ++)
        for (unsigned int j = 0; j < R.cols; j ++)
            for (unsigned int k = 0; k < (cols < M.lins ? cols : M.lins); k ++)
                R.base [i*R.cols + j] += base [i*cols + k] * M.base [k*M.cols + j];
    return (R);
}
//---------------------------------------------------------------------------
Vector operator * (const Matrix & M, const Vector & V) { // multiply a matrix by a vector
    Vector R (M.lins);
    for (unsigned int i = 0; i < R.rows; i ++)
        for (unsigned int k = 0; k < (M.cols < V.rows ? M.cols : V.rows); k ++)
            R.base [i] += M.base [i * M.cols + k] * V.base [k];
    return (R);
}

Vector operator * (const Vector & V, const Matrix & M) { // multiply a vector by a matrix
    Vector R (M.cols);
    for (unsigned int j = 0; j < R.rows; j ++)
        for (unsigned int k = 0; k < (V.rows < M.lins ? V.rows : M.lins); k ++)
            R.base [j] += V.base [k] * M.base [k * M.cols + j];
    return (R);
}
//---------------------------------------------------------------------------
Vector operator * (const Vector & V, const double s) { // multiply a vector by a scalar
    Vector R (V.rows);
    for (unsigned int n = 0; n < R.rows; n ++)
        R.base [n] = V.base [n] * s;
    return (R);
]

Vector operator * (const double s, const Vector & V) { // multiply a scalar by a vector
    Vector R (V.rows);
    for (unsigned int n = 0; n < R.rows; n ++)
        R.base [n] = s * V.base [n];
    return (R);
}
//---------------------------------------------------------------------------
Matrix operator * (const Matrix & M, const double s) { // multiply a matrix by a scalar
    Matrix R (M.lins, M.cols);
    for (unsigned int n = 0; n < R.lins * R.cols; n ++)
        R.base [n] = M.base [n] * s;
    return (R);
}

Matrix operator * (const double s, const Matrix & M) { // multiply a scalar by a matrix
    Matrix R (M.lins, M.cols);
    for (unsigned int n = 0; n < R.lins * R.cols; n ++)
        R.base [n] = s * M.base [n];
    return (R);
]
//---------------------------------------------------------------------------
Vector operator + (const Vector & V, const double s) { // add a vector and a scalar
    Vector R (V.rows);
    for (unsigned int n = 0; n < R.rows; n ++)
        R.base [n] = V.base [n] + s;
    return (R);
}

Vector operator + (const double s, const Vector & V) { // add a scalar and a vector
    Vector R (V.rows);
    for (unsigned int n = 0; n < R.rows; n ++)
        R.base [n] = s + V.base [n];
    return (R);
```

```
}
//---------------------------------------------------------------------------
Matrix operator + (const Matrix & M, const double s) { // add a matrix and a scalar
    Matrix R (M.lins, M.cols);
    for (unsigned int n = 0; n < R.lins * R.cols; n ++)
        R.base [n] = M.base [n] + s;
    return (R);
}

Matrix operator + (const double s, const Matrix & M) { // add a scalar and a matrix
    Matrix R (M.lins, M.cols);
    for (unsigned int n = 0; n < R.lins * R.cols; n ++)
        R.base [n] = s + M.base [n];
    return (R);
}
//---------------------------------------------------------------------------
Vector operator - (const Vector & V, const double s) { // subtract a scalar from a vector
    Vector R (V.rows);
    for (unsigned int n = 0; n < R.rows; n ++)
        R.base [n] = V.base [n] - s;
    return (R);
}

Vector operator - (const double s, const Vector & V) { // subtract a vector from a scalar
    Vector R (V.rows);
    for (unsigned int n = 0; n < R.rows; n ++)
        R.base [n] = s - V.base [n];
    return (R);
}
//---------------------------------------------------------------------------
Matrix operator - (const Matrix & M, const double s) { // subtract a scalar from a matrix
    Matrix R (M.lins, M.cols);
    for (unsigned int n = 0; n < R.lins * R.cols; n ++)
        R.base [n] = M.base [n] - s;
    return (R);
}

Matrix operator - (const double s, const Matrix & M) { // subtract a matrix from a scalar
    Matrix R (M.lins, M.cols);
    for (unsigned int n = 0; n < R.lins * R.cols; n ++)
        R.base [n] = s - M.base [n];
    return (R);
}
//---------------------------------------------------------------------------
```

REFERENCES

ASHRAE, 1989. **Handbook of Fundamentals**, SI Edition. Atlanta, Georgia: ASHRAE.

Axaopoulus, P., P. Panagakis and S. Kyritsis. 1992. Computer simulation assessment of the thermal microenvironment of growing pigs under summer conditions. **Transactions of the ASAE 35** (3): 1005-1009.

Bastianelli, D., D. Sauvant and A. Rérat. 1996. Mathematical modeling of digestion and nutrient absorption in pigs. **Journal of Animal Science 74**: 1873-1887.

Beckett, F. E. 1965. Effective temperature for evaluating or designing hog environments. **Transactions of the ASAE 8** (2): 163-166.

Black, J. L., R. G. Campbell, I. H. Williams, K. J. James and G. T. Davies. 1986. Simulation of energy and amino acid utilisation in the pig. **Research and Development in Agriculture 3** (3): 121-145.

Bond, T. E., C. F. Kelly and H. Heitman Jr. 1959. Hog house air conditioning and ventilation data. **Transactions of the ASAE 2** (1): 1-4.

Bridges, T. C., L. W. Turner, E. M. Smith, T. S. Stahly and O. J. Loewer. 1986. A mathematical procedure for estimating animal growth and body composition. **Transactions of the ASAE 29** (5): 1342-1347.

Bridges, T. C., L. W. Turner, T. S. Stahly, J. L. Usry and O. J. Loewer. 1992a. Modeling the physiological growth of swine. 1. Model logic and growth concepts. **Transactions of the ASAE 35** (3): 1019-1028.

Bridges, T. C., L. W. Turner, J. L. Usry and J. A. Nienaber. 1992b. Modeling the physiological growth of swine. 2. Validation of model logic and growth concepts. **Transactions of the ASAE 35** (3): 1029-1033.

Bruce, J. M. and J. J. Clark. 1979. Models of heat production and critical temperature for growing pigs. **Animal Production 28** (3): 353-369.

Bucklin, R. A., I. A. Nääs, F. S. Zazueta and W. R. Walker. 1991. Natural ventilation in swine housing. Computer Series, Bulletin 270. Gainesville, FL: Florida Cooperative Extension Service, Institute of Food and Agricultural Sciences, University of Florida. 38 p.

Close, W. H. 1978. The effects of plane of nutrition and environmental temperature on the energy metabolism of the growing pig. 3. The efficiency of energy utilization for maintenance and growth. **British Journal of Nutrition 40** (3): 433-438.

Close, W. H. and L. E. Mount. 1978. The effects of plane of nutrition and environmental temperature on the energy metabolism of the growing pig. 1. Heat loss and critical temperature. **British Journal of Nutrition 40** (3): 413-421.

Close, W. H., L. E. Mount and D. Brown. 1978. The effects of plane of nutrition and environmental temperature on the energy metabolism of the growing pig. 2. Growth rate, including protein and fat deposition. **British Journal of Nutrition 40** (3): 423-431.

CRC. 1985. **CRC Handbook of Chemistry and Physics**, 66th Ed. Boca Raton, FL: CRC Press, Inc.

Curtis, S. E. 1983. **Environmental Management in Animal Agriculture**. Ames, Iowa: Iowa State University Press. 410 p.

Ingram, D. L. 1974. Heat loss and its control in pigs. In: Montieth, J. L. and L. E. Mount, eds. **Heat Loss from Animals and Man**. London: Butterworths. p. 233-254.

Ingram, D. L. and L. E. Mount. 1975. **Man and Animals in Hot Environments**. New York: Springer-Verlag. 185 p.

Jacobson, L. D., S. G. Cornelius and K. A. Jordan. 1989. Environmental modifications in a pig growth model for early-weaned piglets. **Animal Production 48** (3): 591-599.

Jones, D. D., W. H. Friday and J. D. Mireley. 1987. Expert system to troubleshoot swine ventilation problems. ASAE Paper No. 87-4038. St. Joseph, MI: ASAE.

Korthals, R. L., G. L. Hahn and J. A. Nienaber. 1994. Evaluation of neural networks as a tool for management of swine environments. **Transactions of the ASAE 37** (4): 1295-1299.

Loewer, O. J., L. W. Turner, N. Gay and R. Muntifering. 1987. Using the concept of physiological age to predict the efficiency of growth in beef animals. **Agricultural Systems 24** (4): 269-289.

Maynard, L. A. 1979. **Animal Nutrition**, 7th Ed. New York: McGraw Hill.

McDonald, T. P. and J. A. Nienaber. 1994. Modeling feed intake in group-penned growing-finishing swine. **Transactions of the ASAE 37** (3): 921-927.

McDonald, T. P., J. A. Nienaber and Y. R. Chen. 1991. Modeling eating behavior of growing-finishing swine. **Transactions of the ASAE 34** (2): 591-596.

McDonald, P., R. A. Edwards, J. F. D. Greenhalgh and C. A. Morgan. 1995. **Animal Nutrition**, 5th Ed. Essex, England: Longman Scientific & Technical. 607 p.

Morrison, S. R., T. E. Bond and H. Heitman Jr. 1967. Skin and lung moisture loss from swine. **Transactions of the ASAE 10** (5): 691-692, 696.

Moughan, P. J. 1985. Sensitivity analysis on a model simulating the digestion and metabolism of nitrogen in the growing pig. **New Zealand Journal of Agricultural Research 28** (4): 463-468.

Moughan, P. J. and W. C. Smith. 1984. Prediction of dietary protein quality based on a model of the digestion and metabolism of nitrogen in the growing pig. **New Zealand Journal of Agricultural Research 27** (4): 501-507.

Moughan, P. J., W. C. Smith and G. Pearson. 1987. Description and validation of a model simulating growth in the pig (20-90 kg liveweight). **New Zealand Journal of Agricultural Research 30** (4): 481-489.

Mount. L. E. 1974. The concept of thermal neutrality. In: Montieth, J. L. and L. E. Mount, eds. **Heat Loss from Animals and Man**. London: Butterworths. p. 425-439.

Mount. L. E. 1975. The assessment of thermal environment in relation to pig production. **Livestock Production Science 2** (4): 381-392.

Nääs, I. A., F. S. Zazueta and R. A. Bucklin. 1990. Microcomputer simulation of heat transfer in tropical swine housing. **Agricultural Mechanization in Asia, Africa and Latin America 21** (4): 38-42.

Nienaber, J. A., T. P. McDonald, G. L. Hahn and Y. R. Chen. 1990a. Eating dynamics of growing-finishing swine. **Transactions of the ASAE 33** (6): 2011-2018.

Nienaber, J. A., G. L. Hahn and L. J. Koong. 1990b. Carcass composition and maintenance of swine as affected by rate of gain. ASAE Paper No. 90-4507. St. Joseph, MI: ASAE.

Nienaber, J. A., T. P. McDonald, G. L. Hahn and Y. R. Chen. 1991. Group feeding behavior of swine. **Transactions of the ASAE 34** (1): 289-294.

Nienaber, J. A., G. L. Hahn, T. P. McDonald and R. L. Korthals. 1996. Feeding patterns and swine performance in hot environments. **Transactions of the ASAE 39** (1): 195-202.

Pomar, C., D. L. Harris and F. Minvielle. 1991a. Computer simulation model of swine production systems. 1. Modeling the growth of young pigs. **Journal of Animal Science 69**: 1468-1488.

Pomar, C., D. L. Harris and F. Minvielle. 1991b. Computer simulation model of swine production systems. 2. Modeling body composition and weight of female pigs, fetal development, milk production, and growth of suckling pigs. **Journal of Animal Science 69**: 1489-1502.

Pomar, C., D. L. Harris, P. Savoie and F. Minvielle. 1991c. Computer simulation model of swine production systems. 3. A dynamic herd simulation model including reproduction. **Journal of Animal Science 69**: 2822-2836.

Schinckel, A. P. and C. F. M. de Lange. 1996. Characterization of growth parameters needed as inputs for pig growth models. **Journal of Animal Science 74**: 2021-2036.

Stombaugh, D. P. and I. A. Stombaugh. 1991. Modeling protein synthesis and deposition during swine growth. **Transactions of the ASAE 34** (6): 2522-2532.

Teter, N. C., J. A. DeShazer and T. L. Thompson. 1973. Operational characteristics of meat animals. Part 1 – Swine. **Transactions of the ASAE 16** (1): 157-159.

Usry, J. L., L. W. Turner, T. S. Stahly, T. C. Bridges and R. S. Gates. 1991. GI tract simulation model of the growing pig. **Transactions of the ASAE 34** (4): 1879-1890.

Usry, J. L., L. W. Turner, T. C. Bridges and J. A. Nienaber. 1992. Modeling the physiological growth of swine. 3. Heat production and interaction with environment. **Transactions of the ASAE 35** (3): 1035-1042.

Van Ouwerkerk, E. N. J. 1992. Modelling the heat balance of pigs at animal and housing levels. In: CIGR. 1992. **Second Report of Working Group on Climatization of Animal Houses**. Ghent, Belgium: Centre for Climatization of Animal Houses, State University of Ghent. 147 p.

Vries, A. G. de and E. Kanis. 1992. A growth model to estimate economic values for food intake capacity in pigs. **Animal Production 55** (2): 241-246.

Watt, D. L., J. A. DeShazer, R. C. Ewan, R. L. Harrold, D. C. Mahan and G. D. Schwab. 1987. NCCISWINE: Housing, nutrition and growth simulation model. **Applied Agricultural Research 2** (4): 218-223.

Whittemore, C. T. 1983. Development of recommended energy and protein allowances for growing pigs. **Agricultural Systems 11** (3): 159-186.

Whittemore, C. T. and Fawcett, R. H. 1974. Model responses of the growing pig to the dietary intake of energy and protein. **Animal Production 19**: 221-231.

Whittemore, C. T. and Fawcett, R. H. 1976. Theoretical aspects of a flexible model to simulate protein and lipid growth in pigs. **Animal Production 22** (1): 87-96.

Xin, H. and J. A. DeShazer. 1992. Feeding patterns of growing pigs at warm constant and cyclic temperatures. **Transactions of the ASAE 35** (1): 319-323.

BIOGRAPHICAL SKETCH

Flávio Bello Fialho was born on the 28th of November of 1966, in Porto Alegre, in the state of Rio Grande do Sul, Brazil. He received his primary education in different schools in Brasília, Porto Alegre, Washington D.C. and New York, and his secondary education at the Colégio Americano, in Porto Alegre. He graduated in Agronomy at the Universidade Federal do Rio Grande do Sul (UFRGS) in Porto Alegre, in 1988.

In 1989 he was hired as a researcher by the Empresa Brasileira de Pesquisa Agropecuária (EMBRAPA) at the National Research Center for Swine and Poultry (CNPSA), in Concórdia, SC, Brazil. He concluded his Master's degree in Animal Science at the School of Agronomy of the Universidade Federal do Rio Grande do Sul in 1991. In August 1993, he entered the Graduate School at the University of Florida, to pursue the degree of Doctor of Philosophy, under the supervision of Dr. Ray A. Bucklin.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Ray A. Bucklin, Chairman
Professor of Agricultural and
Biological Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Pedro S. Zazueta, Cochairman
Professor of Agricultural and
Biological Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Roger A. Nordstedt
Professor of Agricultural and
Biological Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Robert O. Myer
Professor of Animal Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Douglas D. Dankel II
Assistant Professor of Computer and Information Science and Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 1997

Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School